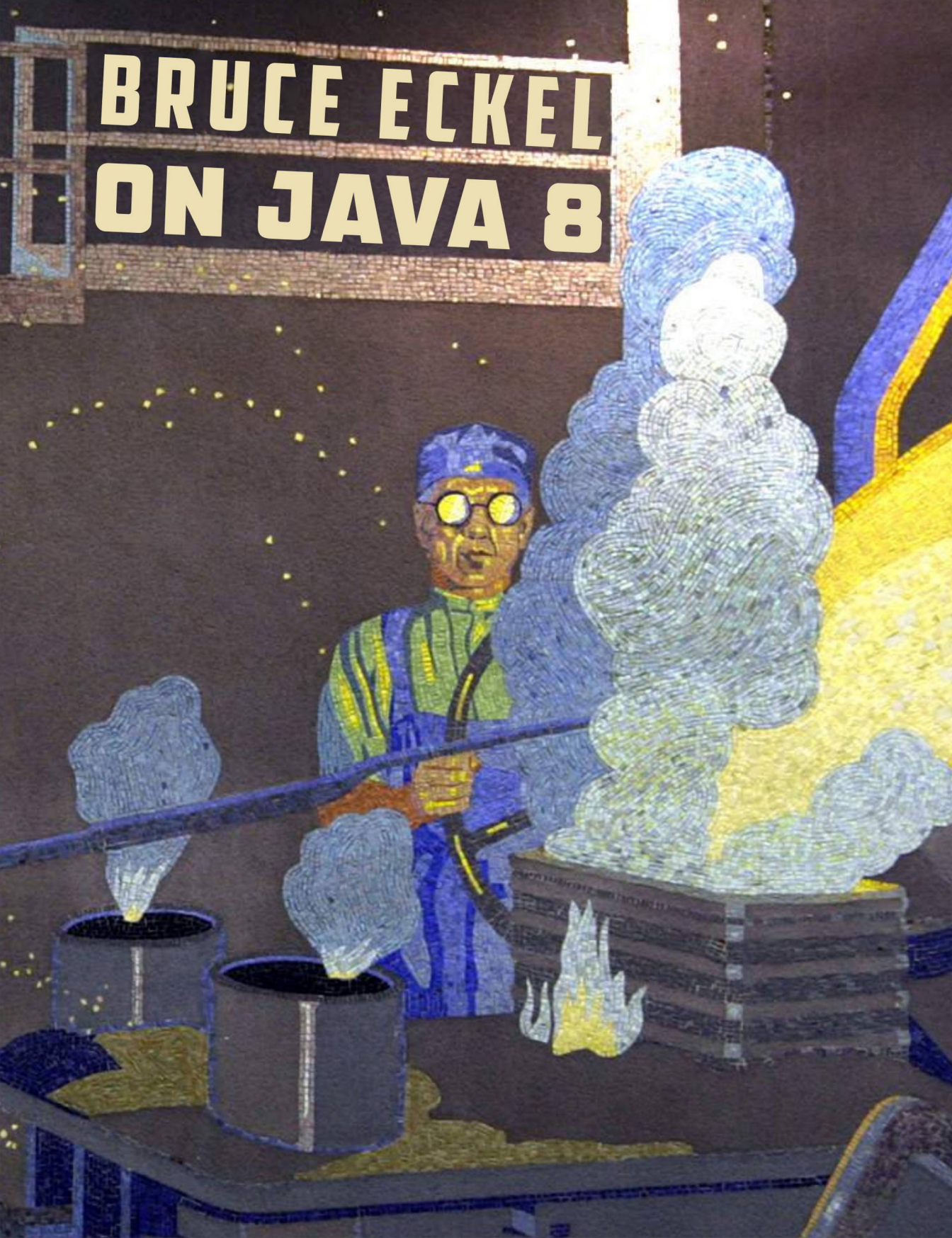
BRUCE ECKEL
ON JAVA 8



On Java 8

Bruce Eckel

This book is for sale at http://leanpub.com/onjava8

This version was published on 2021-12-14

ISBN 978-0-9818725-2-0

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Bruce Eckel

Atomic Kotlin

Contents

About this Sample

This is a free sample of the book *On Java 8*. The English-language version of the book is available in eBook form only at www.OnJava8.com[1].

This sample contains the first few chapters, so you can discover whether it's a good fit for you. Before purchasing the book, you are strongly encouraged to install this sample on your reading device of choice to ensure that you have no problems.

Formats

This sample (and the complete book) is available in multiple eBook formats:

- PDF for reading on computers; includes hyperlinked references to other atoms.
- MOBI for reading on Kindle devices, or the Kindle app.
- EPUB for reading on iPads and other Apple devices (or computer EPUB readers).
- You can also read the book, in your browser, directly from the Leanpub site.

Leanpub provides instructions on how to install an eBook onto your device:

- Apps and Devices for Reading Leanpub Books[2]
- Overview[3]
- Kindle[4]
- iPad/iOS[5]

[1] https://www.OnJava8.com
[2] http://help.leanpub.com/en/articles/3045130-how-can-i-read-a-leanpub-ebook-what-apps-or-devices-do-you-recommend
[3] http://help.leanpub.com/en/articles/117470-after-i-buy-a-leanpub-book-how-can-i-get-it-onto-my-device
[4] http://help.leanpub.com/en/articles/110746-how-do-i-get-leanpub-books-on-my-kindle
[5] http://help.leanpub.com/en/articles/3972511-how-can-i-read-a-leanpub-book-on-apple-s-books-app

The eBook has been tested on these platforms:

- iPad (Open the EPUB file using the Safari browser, then select "Open in iBooks").
- Kindle Fire (See above link for Kindle).
- Kindle Paperwhite (See above link for Kindle).
- Google Play Books[6] (Add the EPUB file to your Books library).

[6] https://play.google.com/books

Copyright

Version 2.0, Copyright December ©2021

Version 1.0, Copyright ©2017

by Bruce Eckel, President, MindView LLC.

ISBN 978-0-9818725-2-0

This book was created as an eBook. That is, it was not first created for print and then converted. It is an eBook first—all layout and formatting is designed to optimize your viewing experience on the various eBook reading platforms and systems.

Cover design by Daniel Will-Harris, www.Will-Harris.com[7].

[7] http://www.Will-Harris.com

Preface

> This book teaches Java programming using the features in the 8th version of that language.

My previous Java book, *Thinking in Java, 4th Edition* (Prentice Hall 2006), is still useful for programming in Java 5, the version of the language used for Android programming. But with the advent of Java 8, the language has changed significantly enough that new Java code feels and reads differently. This justified the two-year effort of creating a new book.

Although there have been other notable improvements to the language since Java 8 (these improvements, through Java 17, are described in the "New Feature:" subsections in this book), Java 8 is still the biggest change to Java since I wrote *Thinking in Java, 4th Edition*. For many programmers and companies, changing to Java 8 is still a challenging next step, and that's who this book is for.

The biggest improvements in Java 8 were added to gain some of the benefits of functional programming. These benefits can be summarized as "lambda expressions, streams, and functional primitives." However, Java was designed as an object-oriented language inspired by Smalltalk. Because of backward compatibility constraints, it cannot suddenly be retrofitted into a functional language. I am deeply impressed by what Brian Goetz and his team have been able to accomplish despite those constraints. Java 8 has definitely made Java better and you will benefit by learning it. I hope this book makes that transition easy and enjoyable for you.

Post-Java-8 Features in this Book

At the time of this writing, Java 17 had just been released. This book was written for Java 8, but upon special request by Turing Book Company (publishers of the Chinese translation of this book), the new features up to and including Java 17 are introduced throughout the book, in special subsections with titles that begin with **New Feature:**,

followed by a description of that feature. These can easily be found in the table of contents. If you are restricted to using Java 8, you can simply skip these subsections without affecting your experience with the rest of the book. New features are *only* used within those subsections, but never in the mainstream Java 8 parts of the book.

The code examples available from the Github repository for this book[8] contain a build file (using the Gradle build tool) designed to work with Java 8. If you install Java 8 (or any subsequent version of Java) on your system, you can successfully run that build. All examples demonstrating new Post-Java-8 features contain a comment at the top of the example that includes the special tag `{NewFeature}` along with the version of the JDK (*Java Development Kit*) which finalized that particular feature (except for the few cases where a nonfinal feature is introduced). The `{NewFeature}` tag automatically excludes that example from the Gradle build. If you want to test the example, you must first install the relevant version of the JDK, at which time you can compile the example using the command-line compiler.

This is Only an eBook

The English version of *On Java 8* is only available as an eBook, and only via www.OnJava8.com[9]. Any other source or delivery mechanism is illegitimate. There is no print version.

Because of the size of this book, my intent has always been to only publish it as an eBook. Color syntax highlighting for code listings is, alone, worth the cost of admission. Searchability, font resizing, text-to-voice for the vision-impaired, the fact you can always keep it with you—there are so many benefits to eBooks it's hard to name them all.

Anyone buying this book needs a computer to run the programs and write code, and the eBook reads nicely on a computer—it even reads tolerably well on a phone. However, the best reading experience is on a tablet computer. Tablets are inexpensive enough that you can now buy one for less than you'd pay for an equivalent print version of this book. It's much easier to read a tablet in bed (for example) than trying to manage the pages of a physical book, especially one this big. When working at your computer, you don't have to hold the pages open when using a tablet at your

[8] https://github.com/BruceEckel/OnJava8-Examples
[9] http://www.OnJava8.com

side. It might feel different at first, but I think you'll find the benefits far outweigh the discomfort of adapting.

Colophon

This book was written with Markdown and produced into various eBook formats using Leanpub[10].

The body font is Linux Libertine and the headline font is Open Sans. The code font is Anonymous Pro Mono, because it is especially compact and allows more characters on a line without wrapping. I chose to place the code inline (rather than make listings into images, as I've seen some books do) because it was important to me that the reader be able to resize the font of the code listings when they resize the body font (otherwise, really, what's the point?).

The build process for the book was automated, as well as the process to extract, compile and test the code examples. All automation was achieved through fairly extensive programs I wrote in Python 3.

Cover Design

The cover of *On Java 8* is from a mosaic created through the *Works Progress Administration* (WPA, a huge project during the US Great Depression from 1935-1943 which put millions of unemployed people back to work). It also reminds me of the illustrations from *The Wizard of Oz* series of books. My friend and designer, Daniel Will-Harris (www.will-harris.com[11]) and I just liked the image.

Acknowledgements

I am grateful for all the benefits from *Thinking in Java*, mostly in the form of speaking engagements throughout the world. These have proved invaluable in creating connections with people and companies.

[10] https://leanpub.com
[11] http://www.will-harris.com

Thanks to Eric Evans (author of *Domain-Driven Design*) for suggesting the book title, and to everyone else in the conference newsgroups for their help in finding the title.

Thanks to James Ward for starting me with the Gradle build tool for this book, and for his help and friendship over the years. Thanks to Ben Muschko for his work polishing the build files, and Hans Dockter for giving Ben the time.

Jeremy Cerise and Bill Frasure came to the developer retreat for the book and followed up with valuable help.

Thanks to all who have taken the time and effort to come to my conferences, workshops, developer retreats, and other events in my town of Crested Butte, Colorado. Your contributions might not be easily seen, but they are deeply important.

Dedication

For my beloved father, E. Wayne Eckel. April 1, 1924—November 23, 2016.

Introduction

> "The limits of my language are the limits of my world"—*Ludwig Wittgenstein (1889-1951)*

This is also true for programming languages. A language can subtly guide you toward certain modes of thought and away from others. Java is particularly opinionated.

Java is a derived language. The original language designers didn't want to use C++ for a project, so they created a new language which unsurprisingly looked a lot like C++, but with improvements. The core changes were the incorporation of a *virtual machine* and *garbage collection*, both of which are described in detail in this book. Java has pushed the industry forward in other ways.

One of the most predominant Java concepts came from the SmallTalk language, which insists that the "object" (described in the next chapter) is the fundamental unit of programming, so everything must be an object. Time has shown this belief to be overenthusiastic. Some folks even declare that objects are a complete failure and should be discarded. Personally, I find making everything an object is not only an unnecessary burden but also pushes many designs in a poor direction. However, there are still situations where objects shine. Requiring that everything be an object (especially all the way down to the lowest level) is a design mistake, but banning objects altogether seems equally draconian.

There are other Java design decisions that haven't panned out as promised. Throughout this book I explain these features so you understand why they don't feel quite right. It's not about declaring that Java is good or bad. If you understand the flaws and limitations of the language you will not get stymied when you encounter a feature that seems "off," and create better code because you know where the boundaries are.

Programming is about managing complexity: the complexity of the problem, laid upon the complexity of the machine. This complexity causes many programming projects to fail.

Language design decisions are often made with complexity in mind, but at some point other issues are considered essential. Those other issues are what cause programmers

to eventually "hit the wall" with a language. For example, C++ had to be backward-compatible with C (to allow easy migration for C programmers), as well as efficient. Those are useful goals and account for much of the success of C++, but they also produce extra complexity.

Visual BASIC (VB) was tied to BASIC, which wasn't really designed as an extensible language. All the extensions piled upon VB have produced some truly un-maintainable syntax. The Perl language is backward-compatible with `awk`, `sed`, `grep`, and other Unix tools it was meant to replace, and as a result it is often accused of producing "write-only code" (that is, you can't read your own code). On the other hand, C++, VB, Perl, and other languages such as SmallTalk had *some* of their design efforts focused on the issue of complexity and as a result are remarkably successful in solving certain types of problems.

The communication revolution enables all of us to connect with each other more easily: one-on-one as well as in groups and as a planet. Perhaps the next revolution is the formation of a kind of global mind resulting from enough people and enough interconnectedness. Java might be one of the tools for that revolution.

Goals

Each chapter teaches a concept, or a group of associated concepts, without relying on features that haven't yet been introduced. That way you can digest each piece in the context of your current knowledge before moving on.

My goals in this book are to:

1. Present the material one step at a time so you can easily incorporate each idea before moving on. Also, to carefully sequence the presentation of features so you're exposed to a topic before you see it in use. This isn't always possible; in those situations, I give a brief introductory description.
2. Use examples that are as simple and short as possible. This sometimes prevents me from tackling "real world" problems, but I've found that readers are usually happier when they can understand every detail of an example rather than being impressed by the scope of the problem it solves. For this I might receive criticism for using "toy examples," but I'm willing to accept that in favor of producing something pedagogically useful.

3. I believe there are some facts that 95 percent of programmers never need to know—details that just confuse people and increase their perception of the complexity of the language.
4. Provide a solid foundation so you understand the issues well enough to move on to more difficult coursework and books.

Language Design Errors

Every language has design errors. New programmers experience deep uncertainty and frustration when they must wade through features and guess at what they should use and what they shouldn't. It's embarrassing to admit mistakes, but this bad beginner experience is a lot worse than the discomfort of acknowledging errors. Alas, every failed language/library design experiment is forever embedded in the Java distribution.

The Nobel laureate economist Joseph Stiglitz has a philosophy of life that applies here, called *The Theory of Escalating Commitment*:

> "The cost of continuing mistakes is borne by others, while the cost of admitting mistakes is borne by yourself."

When I find design errors in a language, I tend to point them out. Java has developed a particularly avid following, folks who treat the language more like a country of origin and less like a programming tool. Because I've written about Java, they assume I am a fellow patriot. When I criticize the errors I find, it tends to have two effects:

1. Initially, a lot of "my-country-right-or-wrong" furor. Eventually—this can take years—the error is acknowledged and seen as just part of the history of Java.
2. More importantly, new programmers don't go through the struggle of wondering why the language was designed this way, especially the self-doubt that comes from finding something that just doesn't seem right, then assuming *I must be doing it wrong or I just don't get it.* Worse, those who teach the language often go right along with the misconceptions rather than delving in and analyzing the issue. By understanding the language design errors, new programmers see mistakes for what they are, and move forward.

Understanding language and library design errors is essential because of their impact on programmer productivity. While seductive on the surface, some features can block your progress when you least expect it. Design errors also inform the creation and adoption of new languages. It's fun to explore what can be done with a language, but design errors tell you what *can't* be done with that language.

For many years, I honestly felt a lack of care from the Java designers regarding their users. Some errors seemed so blatant, so poorly thought-out, that it appeared the designers had some motivation in mind other than serving their users. This seeming lack of respect for programmers is the major reason I moved away from Java and didn't want anything to do with it for such a long time.

When I did start looking into Java again, something about Java 8 felt very different, as if a fundamental shift had occurred in the designers' attitude about the language and its users. Many features and libraries that had been warts on the language were fixed after years of ignoring user complaints. New features felt very different, as if there were now folks on board who were extremely interested in programmer experience. These folks were finally working to make the language better, rather than just quickly adding ideas without delving into their implications. And some of the new features are downright elegant—or as elegant as possible, given Java constraints.

Because of this new focus by the language developers—and I don't think I'm imagining it—writing this book has been dramatically better than past experiences. Java 8 contains fundamental and important improvements. Alas, because of Java's rigid backward-compatibility promise, these improvements required great effort so it's unlikely we'll see anything this dramatic again (I hope I'm wrong about this). Nonetheless, I applaud those who have turned the ship as much as they have and set the language on a better course. For the first time I can recall, I found myself saying "I love that!" about some of the Java code I've been able to write in Java 8.

Popularity

Java's popularity has significant implications. If you learn it, getting a job will probably be easier. There are many training materials, online courses, and other learning resources available. If you're starting a company and you choose to work in Java, it can be easier to find programmers, and that's a compelling argument.

Short-term thinking is almost always a bad idea. Don't use Java if you really don't like it—using it just to get a job is an unhappy life choice. As a company, think hard before choosing Java just because you can hire people. There might be another language that makes fewer employees far more productive (for example, see my book Atomic Kotlin[12]). Using a newer and more exciting language might make it easier to attract programmers, as well.

But if you do enjoy it, if Java *does* call to you, then welcome. I hope this book enriches your programming experience.

Android Programmers

Android programming with Java is tied to Java 5. I've made this book as "Java 8 as possible," so if you want to program in Java for Android devices, you're better off with *Thinking in Java, 4th edition*. There are also many other resources specializing in Android programming.

Note that Kotlin[13] is a newer and much better choice—and an official language—for Android programming.

Book Updates

In 2017, I published this book on Google Play Books. As an experiment, it was *exclusively* on that platform. This not only turned out to be a challenging process, but because Google doesn't do business in a number of countries where I have readers, it prevented those readers from getting the book. It wasn't a good fit, but in the meantime I wrote off the experiment and focused on the next book, Atomic Kotlin[14].

Over time, awareness of the book trickled out and it turned out there was demand for it. After putting *Atomic Kotlin* on Leanpub, I decided it would be worthwhile to convert *On Java 8* to Leanpub as well, which is what you now (virtually) hold in your hands.

[12] https://www.AtomicKotlin.com
[13] https://www.AtomicKotlin.com
[14] https://www.atomickotlin.com/

Leanpub generates multiple formats: PDF, ePub (especially nice on Apple iOS devices via iBooks; there are also ePub readers for most other platforms) and MOBI, for Kindle readers. In addition, there's a version that you can read directly online through the Leanpub site.

From the table of contents you can jump to any part of the book. Note that in the PDF version you must click on the *name* of the chapter or subhead you want to go to, not the page number.

Unfortunately, Leanpub does not support indices (I know of no ebook-creation systems that do), but the ebook allows you to easily search for topics.

The initial updates (*not* chronicled below) were primarily to the code, allowing the examples to run with newer versions of Java, along with more recent versions of the Gradle build tool and JUnit testing libraries.

Note that, although newer versions of Java have introduced a handful of features that are useful, Java 8 contained the most sweeping and dramatic changes to the language since Java 5 (covered in *Thinking in Java, 4th Edition*). After reading *On Java 8* you should be able to understand the features that have been added since.

December 2021: Version 2.0

This version contains all corrections and clarifications added during the development of the Chinese translation of this book, published by the Turing Book Company. This adds a number of new subsections with titles that start with **New Feature:**. You can easily locate these in the table of contents. These new subsections introduce features added to Java since Java 8, all the way through Java 17. These post-Java-8 features are *not* used throughout, because then the book would not work with Java 8. The new features are presented in a standalone manner, and the example comments state the Java version requirement and any special command-line arguments necessary to compile and run those examples.

March 2021 Update

- **Patterns** chapter: Thorough rewrite of most examples & prose. This is the largest change in this update.

- Changed from "no-arg constructor" to "zero-argument constructor" throughout, following modern terminology used in the Java documentation.
- Changed "RTTI" to "Reflection" throughout the book, including changing the **Type Information** chapter name to **Reflection**.
- **Objects Everywhere** chapter: Fixed primitive data type table (superscript values were missing).
- **Operators** chapter: Rewrote "Testing Object Equivalence" section.
- **Implementation Hiding** chapter: Rewrote `onjava/Range.java` and added `onjava/TestRange.java`.
- Various technical, grammar and spelling fixes throughout the book.

January 2021 Update

This was primarily a rewrite of the **Reflection** (formerly **Type Information**) chapter. It also added missing `@Override` annotations throughout the book (because Java doesn't enforce the use of `@Override`). The `/* Output:` sections of all examples were updated, and examples were reformatted to improve readability.

The New Java "Release Cadence"

Java versioning has always been odd: early versions of Java from 1.1 through 1.4 used point releases as major releases. With Java 5, they changed to using whole number releases for major releases.

Java has a new versioning system, also called the *release cadence*. This combines:

1. Putting out a new version every six months, using whole release numbers for each.
2. Including trial features so users can experiment with them and expose problems. The main motivation for the six-month version pacing seems to be quickly discovering problems with these trial features. However, because trial features *are not guaranteed to be permanent*, they can be withdrawn if they don't work out for some reason. You should not depend on trial features.

3. Distinguishing between *Short-Term-Support* (STS) releases and *Long-Term-Support* (LTS) releases. Java 8, 11, and 17 are all LTS releases and the others are STS releases with six-month support periods. Support for STS releases is dropped as soon as the next version comes out. Support for the current LTS release is dropped shortly after the next LTS release comes out, typically within a year (free support from Oracle, that is. OpenJDK support might continue for longer).

Both STS and LTS releases may contain trial features.

There are also different *types* of trial features in each Java release:

- **Experimental**: An early trial of a feature; you can consider these to be approximately %25 complete.
- **Preview**: A feature that is fully specified and implemented, but may still change before it is finalized. You can think of these as beta or even release-candidate features. Sometimes you will see features indicated as "Preview 2," which presumably means they have made some adjustments and want further feedback.
- **Incubating**: This denotes an API or a tool (as opposed to a core language feature) that is not yet part of the Java distribution. To use it you must explicitly fetch it because it won't be part of the standard download. For example, `jshell`, the Java *Read-Evaluate-Print-Loop* (REPL), was an incubating feature in Java 8, but available as part of the standard release since Java 9.

Experimental and preview features are collectively called *nonfinal* features. Nonfinal features are *not* automatically available—you must explicitly enable them either on the command line or through settings in your IDE (*Integrated Development Environment*). A few nonfinal features for Java 17 are introduced later in the book and the demonstration examples contain instructions for compiling these on the command line.

For most companies and programmers, I recommend only moving to the LTS versions, as tracking the STS releases will probably involve significant effort, with questionable benefits for adopting such a short-lived release. If you only upgrade to the LTS releases you will be fine—and you won't worry about the rapid release cadence of the STS versions.

What About User Interfaces?

Graphical user interfaces and desktop programming in Java have had a tumultuous—some would say tragic—history.

The original design goal of the graphical user interface (GUI) library in Java 1.0 was to enable the programmer to build a GUI that looks good on all platforms. That goal was not achieved. Instead, the Java 1.0 *Abstract Windowing Toolkit* (AWT) produced a GUI that looked equally mediocre on all systems. In addition, it was restrictive; you could use only four fonts and you could not access any of the more sophisticated GUI elements in your operating system. The Java 1.0 AWT programming model was also awkward and non-object-oriented. A student in one of my seminars (who had been at Sun during the creation of Java) explained why: The original AWT had been conceived, designed, and implemented in a month. Certainly a marvel of productivity, and an object lesson in why design is important.

The situation improved with the Java 1.1 AWT event model, which took a much clearer, object-oriented approach, along with the addition of JavaBeans, a component programming model (now dead) oriented toward the easy creation of visual programming environments. Java 2 (Java 1.2) finished the transformation away from the old Java 1.0 AWT by essentially replacing everything with the *Java Foundation Classes* (JFC), the GUI portion of which is called "Swing." These are a rich set of JavaBeans that create a reasonable GUI.

Swing wasn't the final GUI library for Java—Sun made a last attempt, called *JavaFX*. When Oracle bought Sun they changed the original ambitious project (which included a scripting language) into a library, and now it appears to be the only UI toolkit getting development effort (see the Wikipedia article on JavaFX)—but even that effort has diminished. JavaFX, too, seems eventually doomed.

Swing is still part of the Java distribution (but it only receives maintenance, no new development), and with Java now an open-source project it should always be available. Also, Swing and JavaFX have some limited interactivity, presumably to aid the transition to JavaFX.

Ultimately, desktop Java never took hold, and never even touched the designers' ambitions. Other pieces, such as JavaBeans, were given much fanfare (many unfortunate authors spent a lot of effort writing books solely on Swing and even books

just on JavaBeans) but never gained any traction. The most usage for desktop Java is for integrated development environments (IDEs) and some in-house corporate applications. People do develop user interfaces in Java, but it's safe to consider this a niche usage of the language.

If you must learn Swing, it's covered in *Thinking in Java, 4th Edition* (available at www.OnJava8.com[15]), and in books dedicated to the topic.

JDK HTML Documentation

The *Java Development Kit* (JDK) from Oracle (a free download[16]) comes with documentation in electronic form, readable through your Web browser. Unless necessary, this book does not repeat that documentation, because it's usually much faster to find the class descriptions with your browser than to look them up in a book (also, the online documentation is current). I'll simply refer to "the JDK documentation." I provide extra descriptions of the classes when the JDK documentation is insufficient.

Tested Examples

The code examples in this book compile with Java 8 and the Gradle[17] build tool. I have also tested it with newer versions of Java, but I recommend you choose one of the *Long-Term Support* versions of the language—at this writing, Java 11 or Java 17. All the examples are in a freely-accessible Github repository[18].

Without a built-in test framework with tests that run every time you build your system, you have no way of knowing whether your code is reliable. To accomplish this in the book, I created a test system to display and validate the output of most examples. At the end of each example that produces output, you'll see the output attached as a block comment. In some cases only the first few lines are shown, or the first and last lines. Embedded output improves the reading and learning experience, and provides yet another way to verify the correctness of the examples.

[15] http://www.OnJava8.com
[16] http://java.oracle.com
[17] https://gradle.org/
[18] https://github.com/BruceEckel/Onjava8-examples/

Coding Standards

In the text of this book, identifiers (keywords, methods, variables, and class names) are set in fixed-width `code font`. Some keywords, such as `class`, are used so much that the special font can become tedious. Terms that are distinctive enough are left in the normal font.

I use a particular coding style for the examples in this book. As much as possible within the book's formatting constraints, this follows the style that Oracle itself uses in virtually all code you find at its site, and seems to be supported by most Java development environments. As the subject of formatting style is good for hours of hot debate, I'll just say I'm not trying to dictate correct style via my examples; I have my own motivation for using the style I do. Because Java is a free-form programming language, continue to use whatever style you're comfortable with. One solution to the coding style issue is to use an IDE (*integrated development environment*) tool like JetBrains IntelliJ IDEA or Visual Studio Code (VSCode) to change formatting to that which suits you.

The code files in this book are tested with an automated system, and should work without compiler errors (except those specifically tagged) in the latest version of Java.

This book focuses on and is tested with Java 8. If you must learn about earlier releases of the language not covered here, you can find the 4th edition of *Thinking in Java* through www.OnJava8.com[19].

Bug Reports

No matter how many tools a writer uses to detect errors, some always creep in and leap off the page for a fresh reader. If you discover something you believe is an error, please submit it along with your suggested correction, for either prose or examples, here[20]. Your help is appreciated.

[19] http://www.OnJava8.com
[20] https://github.com/BruceEckel/Onjava8-examples/issues

Source Code

The source code for this book is available on Github[21]. You may use this code in classroom and other educational situations.

The primary goal of the copyright is to ensure this book is properly cited, and to prevent you from republishing the code without permission. (As long as this book is cited, using examples from the book in most media is generally not a problem.)

In each source-code file you find a reference to the following copyright notice:

```
// Copyright.txt
This computer source code is Copyright ©2021 MindView LLC.
All Rights Reserved.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "On
Java 8" is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView LLC, PO Box 969, Crested Butte, CO 81224
MindViewInc@gmail.com

4. The Source Code and documentation are copyrighted by
MindView LLC. The Source code is provided without express
or implied warranty of any kind, including any implied
```

[21] https://github.com/BruceEckel/Onjava8-examples/

```
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView LLC does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView
LLC makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
and performance of any program that includes the Source
Code is with the user of the Source Code. The user
understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source
Code or any resulting software prove defective, the user
assumes the cost of all necessary servicing, repair, or
correction.

5. IN NO EVENT SHALL MINDVIEW LLC, OR ITS PUBLISHER BE
LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF
BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR
PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE
CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY
TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW LLC, OR
ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE. MINDVIEW LLC SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING
SERVICES FROM MINDVIEW LLC, AND MINDVIEW LLC HAS NO
OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView LLC maintains a Web site which
is the sole distribution point for electronic copies of the
Source Code, https://github.com/BruceEckel/OnJava8-examples,
where it is freely available under the terms stated above.

If you think you've found an error in the Source Code,
please submit a correction at:
```

`https://github.com/BruceEckel/OnJava8-examples/issues`

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

What is an Object?

> "We do not realize what tremendous power the structure of an habitual language has. It is not an exaggeration to say that it enslaves us through the mechanism of semantic reactions and that the structure which a language exhibits, and impresses upon us unconsciously, is automatically projected upon the world around us."—Alfred Korzybski (1930)

The genesis of the computer revolution was in a machine. Our programming languages thus tend to look like that machine.

But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs was fond of saying) and a different kind of expressive medium. As a result, tools are beginning to look less like machines and more like parts of our minds.

Programming languages are the fabric of thought for creating applications. Languages take inspiration from other forms of expression such as writing, painting, sculpture, animation, and filmmaking.

Object-oriented programming (OOP) is one experiment in using the computer as an expressive medium.

Many people feel uncomfortable wading into object-oriented programming without understanding the big picture, so the concepts introduced here give you an overview of OOP. Others might not understand such an overview until they are exposed to the mechanism, becoming lost without seeing code. If you're part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter—skipping it now will not prevent you from writing programs or learning the language. However, come back here eventually to fill in your knowledge so you understand why objects are important and how to design with them.

This chapter assumes you have some programming experience, although not necessarily in C. If you need more preparation in programming before tackling this book, work through the free Thinking in C[22] seminar.

[22] https://archive.org/details/ThinkingInC

The Progress of Abstraction

All programming languages are abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean, "What is it you are abstracting?" Assembly language[23] is a minimal abstraction of the underlying machine. Many so-called "imperative" languages (such as FORTRAN, BASIC, and C) were themselves abstractions of assembly language. Although they were big improvements, their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (the "solution space," the place where you're implementing that solution, such as a computer) and the model of the problem that is actually solved (the "problem space," the place where the problem exists, such as a business). The effort required to perform this mapping, and the fact it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). Prolog casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. Each of these approaches can be a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (Note that some objects don't have problem-space analogs.) The idea is that the program adapts itself to the lingo of the problem by adding new types of objects. When you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before. Thus, OOP describes the problem in terms

[23] https://en.wikipedia.org/wiki/Assembly_language

of the problem, rather than in terms of the computer where the solution will run. There's still a connection, because objects look somewhat like little computers: Each has state and performs operations. This is similar to objects in the real world—they all have characteristics and behaviors.

Alan Kay summarized five basic characteristics of SmallTalk, the first successful object-oriented language and a language that inspired Java. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object**. Think of an object as a fancy variable; it stores data, but you can "make requests", asking it to perform operations on itself. You can usually take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. When you "send a message" to an object, it's a request to call a method that belongs to that object.
3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by packaging existing objects. This hides the complexity of a program behind the simplicity of objects.
4. **Every object has a type**. Each object is an *instance* of a *class*, where "class" is (approximately) synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"
5. **All objects of a particular type can receive the same messages**. This is a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handles anything that fits the description of a shape. This *substitutability* is a foundation of OOP.

Grady Booch offers an even more succinct description of an object:

> *An object has state, behavior and identity*

This means an object can have internal data (which gives it state), methods (to produce behavior), and each object is uniquely distinguished from every other object—that is, every object has a unique address in memory.[24]

[24]This is actually a bit restrictive because objects can conceivably exist in different machines and address spaces, and they can also be stored on disk. In these cases, the identity of the object must be determined by something other than a memory address.

An Object Has an Interface

Aristotle was probably the first to begin a careful study of the concept of *type*; he spoke of "the class of fishes and the class of birds." The idea that all objects, while unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword `class` that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have numerous tellers, customers, accounts, transactions, and units of money—many "objects." Objects that are identical except for their state are grouped together into "classes of objects," and that's where the keyword `class` arose.

Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests*; you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: Every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state: Each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versa.[25]

A class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), so a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs.

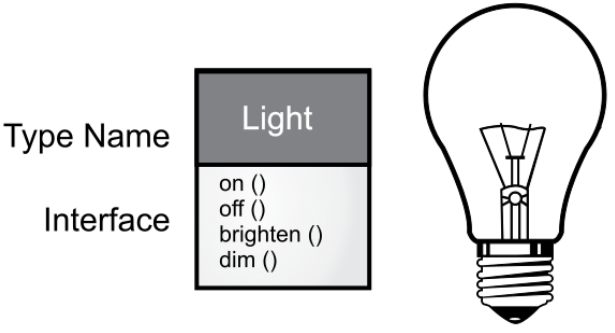
[25]In some cases we make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

The programming system welcomes the new classes and gives them the same care and type checking it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, OOP techniques help reduce a large set of problems to a simpler solution.

Once a class is established, make as many objects of that class as you like, then manipulate those objects as if they are the elements that exist in your problem. Indeed, one of the challenges of object-oriented programming is creating a one-to-one mapping between the elements in the problem space and objects in the solution space.

How do you get an object to do useful work? You make a request of that object—complete a transaction, draw something on the screen, turn on a switch. Each object accepts only certain requests, defined by its *interface* The type determines the interface. As a simple example, consider a representation for a light bulb:




```
Light lt = new Light();
lt.on();
```

The interface determines the requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. A type has a method associated with each possible request, and when you make a particular request to an object, that method is called. This process is usually summarized by saying you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the class is `Light`, the name of this particular `Light` object is `lt`, and the requests you can make of a `Light` object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a `Light` object by defining a "reference" (`lt`) for that object and calling `new` to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The preceding diagram follows the format of the *Unified Modeling Language* (UML). Each class is represented by a box, with the type name in the top portion of the box, any *data members* you care to describe in the middle portion of the box, and the *methods* (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public methods are shown in UML design diagrams, so the middle portion is not shown, as in this case. If you're interested only in the class name, the bottom portion doesn't need to be shown, either.

Objects Provide Services

When trying to develop or understand a program design, an excellent way to think about objects is as "service providers." Your program itself will provide services to the user, and it will accomplish this by using the services offered by other objects. Your goal is to produce (or better, locate in existing code libraries) a set of objects providing the ideal services to solve your problem.

A way to start doing this is to ask, "If I could magically pull them out of a hat, what objects would solve my problem right away?" For example, suppose you are creating a bookkeeping program. You might imagine objects that contain predefined bookkeeping input screens, other objects that perform bookkeeping calculations, and an object that handles printing of checks and invoices on all different kinds of printers. Maybe some of these objects already exist, and for the ones that don't, what would they look like? What services would those objects provide, and what objects would they need to fulfill their obligations? If you keep doing this, you eventually reach a point where you say either, "That object seems simple enough to sit down and write" or "I'm sure that object must exist already." This is a reasonable way to decompose a problem into a set of objects.

Thinking of an object as a service provider has an additional benefit: It helps improve the cohesiveness of the object. *High cohesion* is a fundamental quality of software design: It means the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) "fit together" well. One problem people have when designing objects is cramming too much functionality into one object. For example, in your check printing module, you might decide you need an object that knows all about formatting and printing. You'll probably discover this is too much for one object, and that what you need is three or more objects. One object might be a catalog of all possible check layouts, which can be queried for information about how to print a check. One object or set of objects can be a generic printing interface that knows all about different kinds of printers (but nothing about bookkeeping—perhaps a candidate for buying instead of creating). A third object uses the services of the other two to accomplish the task. Thus, each object has a cohesive set of services it offers. In good object-oriented design, each object does one thing well, but doesn't try to do too much. This not only discovers objects that might be procured from another vendor (such as the printer interface object), but it also produces new objects that might be reused somewhere else (such as the catalog of check layouts).

Treating objects as service providers is useful not only during the design process, but also when someone else is trying to understand your code or reuse an object. If they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

The Hidden Implementation

We can break up the playing field into *class creators* (those who create new data types) and *client programmers*[26] (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't access it, which means the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the

[26] I'm indebted to my friend Scott Meyers for this term.

tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs.

All relationships need boundaries, respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library. If all members of a class are available to everyone, the client programmer can do anything with that class and there's no way to enforce rules. Even though you might prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts necessary for the internal operation of the data type but not part of the interface that users need to solve their particular problems. This is actually a service to client programmers because they can easily see what's important and what they can ignore. (Notice this is also a philosophical decision. Some programming languages assume that if a programmer wishes to access the internals, they should be allowed.)

The second reason for access control is to enable the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, then later discover you must rewrite it to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

Java has three explicit keywords to set the boundaries in a class: `public`, `private`, and `protected`. These *access specifiers* determine who can use the definitions that follow. `public` means the element is available to everyone. `private` means no one can access that element except you, the creator of the type, inside methods of that type. `private` is a brick wall between you and the client programmer. Anyone trying to access a `private` member gets a compile-time error. `protected` acts like `private`, with the exception that an inheriting class may access `protected` members, but not `private` members. Inheritance is introduced shortly.

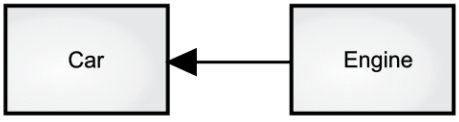
Java also has a "default" access, which comes into play if you don't use one of the aforementioned specifiers. This is usually called *package access* because classes can access the members of other classes in the same `package` (library component), but

outside the package those same members appear to be `private`.

Reusing the Implementation

Once a class is tested, it should (ideally) represent a useful unit of code. This reusability is not nearly so easy to achieve as many hope; it takes experience and insight to produce a reusable object design. But once you have such a design, it begs for reuse. Code reuse is an argument for object-oriented programming languages.

The simplest way to reuse a class is to use an object of that class directly, but you can also place an object of that class inside a new class. Your new class can be made up of any number and type of other objects, in any combination, to produce the desired functionality. Because you *compose* a new class from existing classes, this concept is called *composition* (if composition is dynamic, it's usually called *aggregation*). Composition is often called a *has-a* relationship, as in "A car has an engine."



(This diagram indicates composition with the filled diamond, which states there is one car. I typically use a simpler form: just a line, without the diamond, to indicate an association.[27]

Composition comes with a great deal of flexibility. The member objects of your new class are typically private, making them inaccessible to client programmers who use the class. This means changing those members doesn't disturb existing client code. You can also change the member objects at runtime, to dynamically change the behavior of your program. Inheritance, described next, does not have this flexibility because the compiler must place compile-time restrictions on classes created using inheritance.

Inheritance is is often highly emphasized in object-oriented programming. A new programmer can get the impression that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, first look to

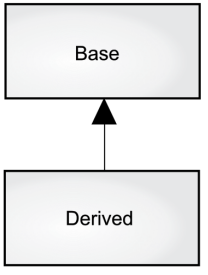
[27]This is enough detail for most diagrams, and you don't need to get specific about whether you're using aggregation or composition.

composition when creating new classes because it is simpler, more flexible, and produces cleaner designs. Once you've had some experience, it is reasonably obvious when you need inheritance.

Inheritance

By itself, the idea of an object is a convenient tool. Objects package data and functionality together by *concept* and represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the `class` keyword.

It seems a pity, however, to go to all the trouble to create a class, then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base class* or *superclass* or *parent class*) is changed, the modified "clone" (called the *derived class* or *inherited class* or *subclass* or *child class*) also reflects those changes.



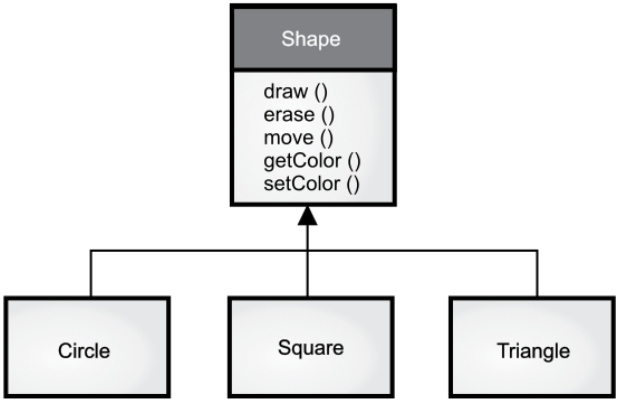
The arrow in this diagram points from the derived class to the base class. As you will see, there is commonly more than one derived class.

A type does more than describe the constraints on a set of objects; it also relates to other types. Two types can have characteristics and behaviors in common, but one type might contain more characteristics than another and might also handle more messages (or handle them differently). Inheritance expresses this similarity through the concept of base types and derived types. A base type contains all characteristics

and behaviors shared among the types derived from it. You create a base type to represent the core of your ideas. From the base type, you derive other types to express the different ways this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash." Each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived with additional characteristics (a bottle has a color, a steel can is magnetic) or behaviors (you can crush an aluminum can). In addition, some behaviors can be different (the value of paper depends on its type and condition). Using inheritance, you build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the common "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited)—circle, square, triangle, and so on—each of which can have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors might be different, such as when you calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



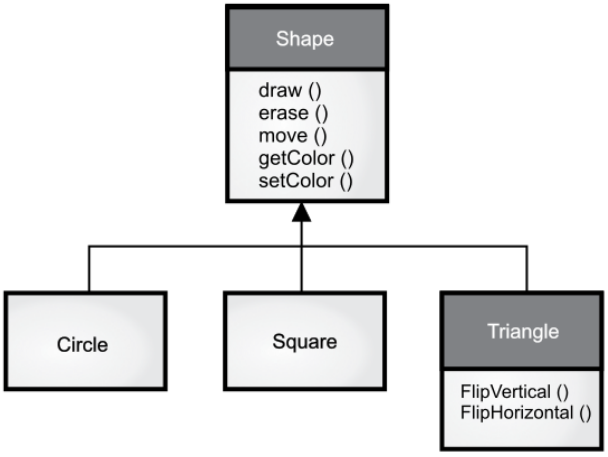
Casting the solution in the same terms as the problem is useful because you don't need intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is an important aspect of the model, so you go directly from the description of the system in the real world to the description

of the system in code. Indeed, sometimes people who are trained to look for complex solutions have difficulty with the simplicity of object-oriented design.

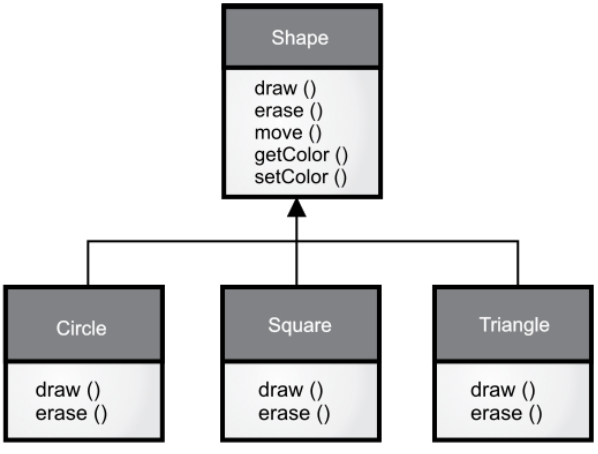
Inheriting from an existing type creates a new type. This new type contains not only all the members of the existing type (although the `private` ones are hidden away and inaccessible), but more importantly it duplicates the interface of the base class. That is, all messages accepted by base-class objects are also accepted by derived-class objects. We know the type of a class by the messages it accepts, so the derived class *is the same type as the base class*. In the previous example, "A circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Because both base class and derived class have the same fundamental interface, there must be some implementation to go along with that interface. That is, there must be executable code when an object receives a particular message. If you inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

There are two ways to differentiate your new derived class from the original base class. The first is straightforward: add brand new methods to the derived class. These new methods are not part of the base-class interface. This means the base class didn't do as much as you wanted, so you added more methods. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, look closely for the possibility that your base class might also need these additional methods (or that you should be using composition instead). This process of discovery and iteration of your design happens regularly in object-oriented programming.



Although inheritance can sometimes imply (especially in Java, where the keyword for inheritance is `extends`) that you are going to add new methods to the interface, that's not necessarily true. The second and more important way to differentiate your new class is to change the behavior of an existing base-class method. This is called *overriding* that method.



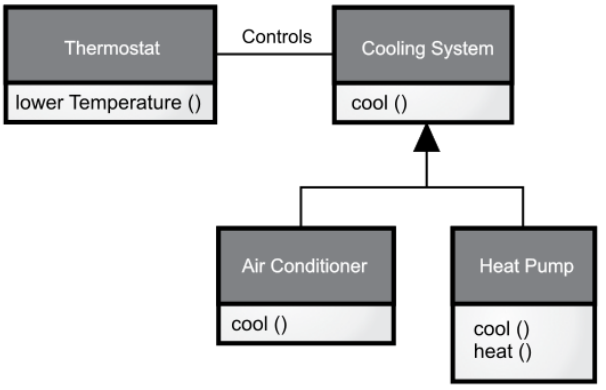
To override a method, you create a new definition for the method in the derived class. You're saying, "I'm using the same interface method here, but I want it to do something different for my new type."

Is-a vs. Is-Like-a Relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class methods (and not add new methods that aren't in the base class)? This would mean that the derived class is *exactly* the same type as the base class because it has exactly the same interface. As a result, you can perfectly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often called the *substitution principle*[28]. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say, "A circle *is a* shape." A test for inheritance is to see if the is-a relationship makes sense for your classes.

Sometimes you add new interface elements to a derived type, thus extending the interface. The new type can still substitute for the base type, but the substitution isn't perfect because your new methods are not accessible from the base type. This can be described as an *is-like-a* relationship (my term). The new type has the interface of the old type but it also contains other methods, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object is extended, and the existing system only knows about the original interface.

[28]Or *Liskov Substitution Principle*, after Barbara Liskov who first described it.



Once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so it can also include heating—at which point the substitution principle will work. However, this diagram shows what can happen with design in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it *is* nice if your design works out that way. But you'll find there are times when it's equally clear you must add new methods to the interface of a derived class (extension). With inspection both cases should be reasonably obvious.

Interchangeable Objects with Polymorphism

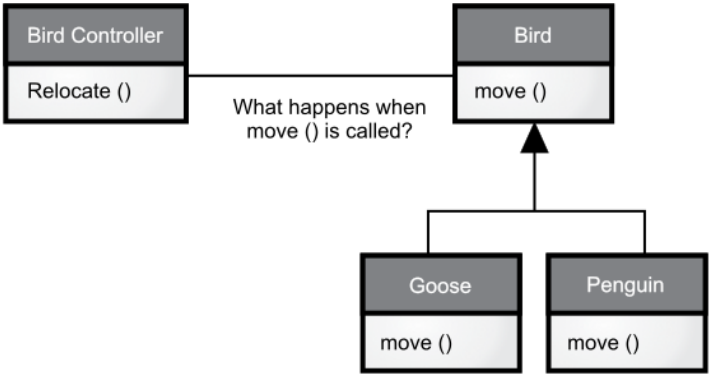
When dealing with type hierarchies, you often treat an object not as the specific type it is, but as its base type. This way you can write code that doesn't depend on specific types. In the shape example, methods manipulate generic shapes, unconcerned about whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these methods send a message to a shape object without worrying how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is a common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called "pentagon" without modifying methods that deal only with generic shapes. This ability to easily extend a

design by deriving new subtypes is one of the essential ways to encapsulate change. This improves designs while reducing the cost of software maintenance.

There's a problem when attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a method tells a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile time precisely what piece of code is executed. That's the whole point—when the message is sent, the programmer doesn't *want* to know what piece of code is executed; the draw method can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type.

If you don't need to know what piece of code is executed, when you add a new subtype, the code it executes can be different without requiring changes to the code that calls it. But what does the compiler do when it cannot know precisely what piece of code is executed? For example, in the following diagram the `BirdController` object just works with generic `Bird` objects and does not know what exact type they are. This is convenient from `BirdController`'s perspective because it doesn't require special code to determine the exact type of `Bird` it's working with or that `Birds` behavior. So how does it happen that, when `move()` is called while ignoring the specific type of `Bird`, the right behavior will occur (a `Goose` walks, flies, or swims, and a `Penguin` walks or swims)?



The answer is the fundamental trick of inheritance: The compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler produces what is called *early binding*, a term you might not have

heard because you've never thought about it any other way. It means the compiler generates a call to a specific function name, which resolves to the absolute address of the code to be executed. With inheritance, the program cannot determine the address of the code until runtime, so some other scheme is necessary when a message is sent to an object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code called isn't determined until runtime. The compiler does ensure that the method exists and performs type checking on the arguments and return value, but it doesn't know the exact code to execute.

To perform late binding, Java uses a special bit of code in lieu of the absolute call. This code calculates the address of the method body, using information stored in the object (this process is covered in great detail in the Polymorphism chapter). Thus, each object behaves differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In some languages you must explicitly grant a method the flexibility of late-binding properties. For example, C++ uses the `virtual` keyword. In such languages, methods are *not* dynamically bound by default. In Java, dynamic binding is the default behavior and you don't need extra keywords to produce polymorphism.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we write a single piece of code that ignores specific details of type and talks only to the base class. That code is *decoupled* from type-specific information and thus is simpler to write and easier to understand. And, if a new type—a `Hexagon`, for example—is added through inheritance, code works just as well for the new type of `Shape` as it did on the existing types. Thus, the program is *extensible*.

If you write a method in Java (you will soon learn how):

```
void doSomething(Shape shape) {
  shape.erase();
  // ...
  shape.draw();
}
```

This method speaks to any `Shape`, so it is independent of the specific type of object

it's drawing and erasing. If some other part of the program uses the `doSomething()` method:

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

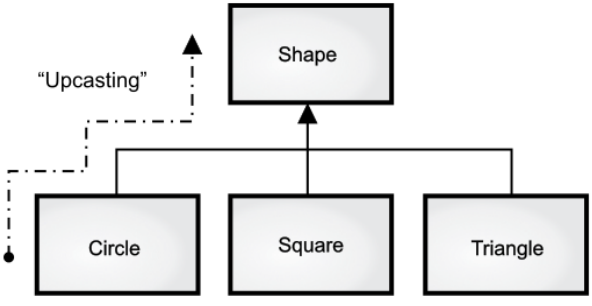
The calls to `doSomething()` automatically work correctly, regardless of the exact type of the object.

This is a rather amazing trick. Consider the line:

```
doSomething(circle);
```

What's happening here is that a `Circle` is passed into a method that expects a `Shape`. Because a `Circle` *is* a `Shape` it is treated as such by `doSomething()`. That is, any message that `doSomething()` can send to a `Shape`, a `Circle` can accept. It is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "upcasting."



An object-oriented program contains upcasting somewhere, because that's how you decouple yourself from knowing the exact type you're working with. Look at the code in `doSomething()`:

```
shape.erase();
// ...
shape.draw();
```

Notice it doesn't say, "If you're a `Circle`, do this, if you're a `Square`, do that, etc." If you write that kind of code, which checks for all the possible types a `Shape` can actually be, it's messy and you must change it every time you add a new kind of `Shape`. Here, you just say, "You're a shape, I know you can `erase()` and `draw()` yourself, do it, and take care of the details correctly."

What's impressive about the code in `doSomething()` is that, somehow, the right thing happens. Calling `draw()` for `Circle` causes different code to be executed than calling `draw()` for a `Square` or a `Line`, but when the `draw()` message is sent to an anonymous `Shape`, the correct behavior occurs based on the actual type of the `Shape`. This is amazing because when the Java compiler is compiling the code for `doSomething()`, it cannot know exactly what types it is dealing with. Ordinarily, you'd expect it to end up calling the version of `erase()` and `draw()` for the base class `Shape`, and not for the specific `Circle`, `Square`, or `Line`. And yet the right thing happens—that's polymorphism. The compiler and runtime system handle the details; all you must know is it happens, and more importantly, how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.

The Singly-Rooted Hierarchy

An OOP issue that has become especially prominent since the introduction of C++ is whether all classes should by default be inherited from a single base class. In Java (as with virtually all other OOP languages *except* for C++) the answer is yes, and the name of this ultimate base class is simply `Object`.

There are many benefits to a singly-rooted hierarchy. All objects have a common interface, so they are all ultimately the same fundamental type. The alternative (provided by C++) is that you don't know that everything is the same basic type. From a backward-compatibility standpoint this fits the model of C better and can be thought of as less restrictive, but for full-on object-oriented programming you must build your own hierarchy to provide the same convenience that's built into other OOP languages. And in any new class library you acquire, some other incompatible

interface is used. It requires effort to work the new interface into your design. Is the extra "flexibility" of C++ worth it? If you need it—if you have a large investment in C—it's quite valuable. If you're starting from scratch, alternatives such as Java can be more productive.

A singly rooted hierarchy makes it much easier to implement a *garbage collector*, one of the fundamental improvements of Java over C++. And since information about the type of an object is guaranteed to be in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system-level operations, such as *exception handling* (a language mechanism for reporting errors), and to allow greater flexibility in programming.

Collections

In general, you don't know how many objects you need to solve a particular problem, or how long they will last. You also don't know how to store those objects. How can you know how much space to create if that information isn't known until runtime?

The solution to most problems in object-oriented design seems flippant: You create another type of object. The new type of object that solves this particular problem holds references to other objects. You can also do the same thing with an *array*, available in most languages. But this new object, generally called a *collection* (also called a *container*, but the Java libraries use "collection" almost universally), will expand itself whenever necessary to accommodate everything you place inside it. You don't need to know how many objects you're going to hold in a collection—just create a collection object and let it take care of the details.

Fortunately, a good OOP language comes with a set of collections as part of the package. In C++, it's part of the Standard C++ Library and is often called the *Standard Template Library* (STL). SmallTalk has a very complete set of collections. Java also has numerous collections in its standard library. In some libraries, one or two generic collections is considered good enough for all needs, and in others (Java, for example) the library has different types of collections for different needs: several different kinds of `List` classes (to hold sequences), `Maps` (also known as *associative arrays*, to associate objects with other objects), `Sets` (to hold one of each type of object), and more components such as queues, trees, stacks, etc.

From a design standpoint, all you really want is a collection you can manipulate to solve your problem. If a single type of collection satisfied all of your needs, we wouldn't need different kinds. There are two reasons you need a choice of collections:

1. Collections provide different types of interfaces and external behavior. Stacks and queues are different from sets and lists. One of these might provide a more flexible solution to your problem than another.
2. Different implementations have different efficiencies for certain operations. For example, there are two basic types of `List`: `ArrayList` and `LinkedList`. Both are simple sequences that can have identical interfaces and external behaviors. But some operations have significantly different costs. Randomly accessing elements in an `ArrayList` is a constant-time operation; it takes the same amount of time regardless of the element you select. However, in a `LinkedList` it is expensive to move through the list to randomly select an element, and it takes longer to find an element that is farther down the list. On the other hand, to insert an element in the middle of a sequence, it's cheaper in a `LinkedList` than in an `ArrayList`. These and other operations have different efficiencies depending on the underlying structure of the sequence. You might start building your program with a `LinkedList` and, when tuning for performance, change to an `ArrayList`. Because of the abstraction via the interface `List`, you can change from one to the other with minimal impact on your code.

Parameterized Types (Generics)

Before Java 5, collections held the one universal type in Java: `Object`. The singly rooted hierarchy means everything is an `Object`, so a collection that holds `Objects` can hold anything.[29] This made collections easy to reuse.

To use such a collection, you add object references to it and later ask for them back. But the collection holds only `Objects`, so when you add an object reference into the collection it is upcast to `Object`, thus losing its character. When fetching it back, you get an `Object` reference, and not a reference to the type you put in. How do you turn it back into something with the specific type of the object you put into the collection?

Here, the cast is used again, but this time you're not casting up the inheritance hierarchy to a more general type. Instead, you cast down the hierarchy to a more

[29] They do not hold primitives, but *autoboxing* simplifies this restriction somewhat. This is discussed in detail later in the book.

specific type, so this manner of casting is called *downcasting*. With upcasting, you know that a `Circle` is a type of `Shape` so it's safe to upcast, but you don't know that an `Object` is necessarily a `Circle` or a `Shape` so it's not safe to downcast unless you determine extra type information about that object.

It's not completely dangerous because if you downcast to the wrong type you'll get a runtime error called an *exception*, described shortly. When you fetch `Object` references from a collection, however, you need some way to remember exactly what they are in order to perform a proper downcast.

Downcasting and the associated runtime checks require extra time for the running program and extra effort from the programmer. Wouldn't it make sense to somehow create the collection so it knows the types it holds, eliminating the need for the downcast and a possible mistake? The solution is called a *parameterized type* mechanism. A parameterized type is a class that the compiler can automatically customize to work with particular types. For example, with a parameterized collection, the compiler can customize that collection so it accepts only `Shapes` and fetches only `Shapes`.

Java 5 added parameterized types, called *generics*, which is a major feature. You'll recognize generics by the angle brackets with types inside; for example, you can create an `ArrayList` to hold `Shape` like this:

```
ArrayList<Shape> shapes = new ArrayList<>();
```

There have also been changes to many of the standard library components to take advantage of generics. You will see that generics have an impact on much of the code in this book.

Object Creation & Lifetime

One critical issue when working with objects is the way they are created and destroyed. Each object requires resources, most notably memory, to exist. When an object is no longer needed it must be cleaned up so these resources are released for reuse. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: You create the object, use it for as long as it's needed, then it should be destroyed. However, it's not hard to encounter situations that are more complex.

Suppose, for example, you are designing a system to manage air traffic for an airport. (The same model might also work for managing crates in a warehouse, or a video rental system, or a kennel for boarding pets.) At first it seems simple: Make a collection to hold airplanes, then create a new airplane and place it in the collection for each airplane that enters the air-traffic-control zone. For cleanup, simply clean up the appropriate airplane object when a plane leaves the zone.

But suppose you have some other system to record data about the planes; perhaps data that doesn't require such immediate attention as the main controller function. Maybe it's a record of the flight plans of all the small planes that leave the airport. So you have a second collection of small planes, and whenever you create a plane object you also put it in this second collection if it's a small plane. Then some background process performs operations on the objects in this collection during idle moments.

Now the problem is more difficult: How can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise in a number of other situations, and in programming systems (such as C++) where you must explicitly delete an object this can become quite complex.

Where is the data for an object and how is the lifetime of the object controlled? C++ takes the approach that efficiency is the most important issue, so it gives the programmer a choice. For maximum runtime speed, the storage and lifetime can be determined while the program is written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, and this control can be very valuable in certain situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program. If you are trying to solve a more general problem such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the *heap*. In this approach, you don't know until runtime how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap when you need it. Because the storage is managed dynamically, at runtime, the amount of time required to allocate storage on the heap can be longer than the time to create storage on the stack (but not necessarily). Creating storage on

the stack is often a single assembly instruction to move the stack pointer down and another to move it back up. The time to create heap storage depends on the design of the storage mechanism.

The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve general programming problems.

Java uses dynamic memory allocation, exclusively.[30] Every time you create an object, you use the `new` operator to build a dynamic instance of that object.

There's another issue, however, and that's the lifetime of an object. With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and automatically destroys it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don't do it correctly. Java is built upon a *garbage collector* which automatically discovers when an object is no longer in use and releases it. A garbage collector is much more convenient because it reduces the number of issues you must track and the code you must write. Thus, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks, which has brought many a C++ project to its knees.

With Java, the garbage collector is designed to take care of the problem of releasing memory (although this doesn't include other aspects of cleaning up an object). The garbage collector "knows" when an object is no longer in use, and automatically releases the memory for that object. This, combined with the fact that all objects are inherited from the single root class `Object` and you can create objects only one way—on the heap—makes the process of programming in Java much simpler than programming in C++. You have far fewer decisions to make and hurdles to overcome.

Exception Handling: Dealing with Errors

Since the beginning of programming languages, error handling has been especially difficult. Because it's so hard to design a good error-handling scheme, many languages ignore the issue, passing the problem on to library designers who come up

[30] Primitive types, which you'll learn about later, are a special case.

with halfway measures that work in many situations but can easily be circumvented, generally by just ignoring errors. A major problem with most error-handling schemes is that they rely on programmers to follow an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant—often the case if they are in a hurry—these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate exception handler designed for that particular type of error. It's as if exception handling is a different, parallel path of execution, taken when things go wrong. Because it uses a separate execution path, it doesn't interfere with your normally executing code. This can make that code simpler to write because you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value returned from a method or a flag set by a method to indicate an error condition—these can be ignored. An exception cannot be ignored, so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting the program, you are sometimes able to set things right and restore execution, which produces more robust programs.

Java's exception handling stands out among programming languages because it was wired in from the beginning and you're forced to use it. It is the single acceptable way to report errors. If you don't write your code to properly handle exceptions, you'll get a compile-time error message. This guaranteed consistency can sometimes make error handling much easier.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented by an object. Exception handling existed before object-oriented languages.

Summary

A procedural program contains data definitions and function calls. To find the meaning of such a program, you must work at it, looking through the function calls and low-level concepts to create a model in your mind. This is the reason we need intermediate representations when designing procedural programs—by themselves,

these programs tend to be confusing because the terms of expression are oriented more toward the computer than to the problem you're solving.

Because OOP adds many new concepts on top of what you find in a procedural language, your natural assumption might be that the resulting Java program is far more complicated than the equivalent procedural program. Here, you'll be pleasantly surprised: A well-written Java program is generally simpler and easier to understand than a procedural program. What you'll see are the definitions of the objects that represent concepts in your problem space (rather than the issues of the computer representation) and messages sent to those objects to indicate activities in that space. One of the delights of object-oriented programming is that, with a well-designed program, it's easy to understand the code by reading it. Usually, there's a lot less code as well, because many problems are solved by reusing existing library code.

OOP and Java might not be for everyone. It's important to evaluate your own needs and decide whether Java will optimally satisfy those needs, or if you might be better off with another programming system (perhaps the one you're currently using). If your needs are very specialized for the foreseeable future and you have specific constraints that might not be satisfied by Java, you owe it to yourself to investigate the alternatives (in particular, I recommend looking at Python[31]). If you still choose Java as your language, you'll at least understand what the options were and have a clear vision of why you took that direction.

[31] https://www.python.org/

Installing Java and the Book Examples

In which we provision ourselves for the journey.

Before you can begin learning the language, you must install Java and the book's source-code examples. Because it is possible for a "dedicated beginner" to learn programming from this book, I explain the process in detail, assuming you haven't previously used the computer's command-line shell. If you have, you can skip forward to the installation instructions.

If any terminology or processes described here are still not clear to you, you can usually find explanations or answers through Google[32]. For more specific issues or problems, try StackOverflow[33]. Sometimes you can find installation instructions on YouTube[34].

Editors

To create and modify Java program files—the code listings shown in this book—you need a program called an *editor*. You'll also need an editor to make changes to your system configuration files, which is sometimes required during installation.

Programming editors vary from full-featured *Integrated Development Environments* (IDEs like IntelliJ IDEA or VSCode) to more basic text manipulation applications. If you already have an IDE and are comfortable with it, feel free to use that for this book, but if you haven't chosen one, try VSCode. Find it here[35].

VSCode is free and open-source, is very simple to install, works on all platforms (Windows, Mac and Linux), and has a built-in Java mode that is automatically

[32] https://www.google.com/
[33] http://stackoverflow.com/
[34] https://www.youtube.com/
[35] https://code.visualstudio.com/

invoked when you open a Java file. It has some handy editing features that you'll probably come to love. More details are on their site.

There are many other editors; these are a subculture unto themselves and people even get into heated arguments about their merits. If you find one you like better, it's not too hard to change. The important thing is to choose one and get comfortable with it.

Although VSCode is a great way to get started, JetBrains IntelliJ IDEA[36] contains more powerful programming features that you might want to explore as you start doing more with Java. I use both editors on a regular basis.

The Shell

If you haven't programmed before, you might be unfamiliar with your operating system *shell* (also called the *command prompt* in Windows). The shell harkens back to the early days of computing when everything happened by typing commands and the computer responded by displaying responses; it was all text-based.

Although it can seem primitive in the age of graphical user interfaces, a shell provides a surprising number of valuable features. We'll use the shell regularly in this book, both as part of the installation process and to run Java programs.

Starting a Shell

Mac: Click on the *Spotlight* (the magnifying-glass icon in the upper-right corner of the screen) and type "terminal." Click on the application that looks like a little TV screen (you might also be able to hit "Return"). This starts a shell in your home directory.

Windows: First, start the Windows Explorer to navigate through your directories:

- *Windows 7*: click the "Start" button in the lower left corner of the screen. In the Start Menu search box area type "explorer" then press the "Enter" key.
- *Windows 8*: click Windows+Q, type "explorer" then press the "Enter" key.
- *Windows 10*: click Windows+E.

[36] https://www.jetbrains.com/idea/

Once the Windows Explorer is running, move through the folders on your computer by double-clicking on them with the mouse. Navigate to the desired folder. Now click the file tab at the top left of the Explorer window and select "Open command prompt." This opens a shell in the destination directory.

Linux: To open a shell in your home directory:

- *Debian*: Press Alt+F2. In the dialog that pops up, type 'gnome-terminal'
- *Ubuntu*: Either right-click on the desktop and select 'Open Terminal', or press Ctrl+Alt+T
- *Redhat*: Right-click on the desktop and select 'Open Terminal'
- *Fedora*: Press Alt+F2. In the dialog that pops up, type 'gnome-terminal'

Directories

Directories are one of the fundamental elements of a shell. Directories hold files, as well as other directories. Think of a directory as a tree with branches. If `books` is a directory on your system and it has two other directories as branches, for example `math` and `art`, we say you have a directory `books` with two *subdirectories* `math` and `art`. We refer to them as `books/math` and `books/art` because `books` is their *parent* directory. Note that Windows uses backslashes rather than forward slashes to separate the parts of a directory.

Basic Shell Operations

The shell operations I show here are approximately identical across operating systems. For the purposes of this book, here are the essential operations in a shell:

- **Change directory**: Use `cd` followed by the name of the directory where you want to move, or `cd ..` if you want to move up a directory. If you want to move to a different directory while remembering where you came from, use `pushd` followed by the different directory name. Then, to return to the previous directory, just say `popd`.
- **Directory listing**: `ls` (`dir` in Windows) displays all the files and subdirectory names in the current directory. Use the wildcard `*` (asterisk) to narrow your search. For example, if you want to list all the files ending in ".java," you say `ls *.java` (Windows: `dir *.java`). If you want to list the files starting with "F" and ending in ".java," you say `ls F*.java` (Windows: `dir F*.java`).

- **Create a directory**: use the `mkdir` ("make directory") command (Windows: `md`), followed by the name of the directory you want to create. For example, `mkdir books` (Windows: `md books`).
- **Remove a file**: Use `rm` ("remove") followed by the name of the file you wish to remove (Windows: `del`). For example, `rm somefile.java` (Windows: `del somefile.java`).
- **Remove a directory**: use the `rm -r` command to remove the files in the directory and the directory itself (Windows: `deltree`). For example, `rm -r books` (Windows: `deltree books`).
- **Repeat a command**: The "up arrow" on all three operating systems moves through previous commands so you can edit and repeat them. On Mac/Linux, `!!` repeats the last command and `!n` repeats the nth command.
- **Command history**: Use `history` in Mac/Linux or press the F7 key in Windows. This gives you a list of all the commands you've entered. Mac/Linux provides numbers to refer to when you want to repeat a command.
- **Unpacking a zip archive**: A file name ending with `.zip` is an archive containing other files in a compressed format. Both Linux and Mac have command-line `unzip` utilities, and you can install a command-line `unzip` for Windows via the Internet. However, in all three systems the graphical file browser (Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux) will browse to the directory containing your zip file. Then right-mouse-click on the file and select "Open" on the Mac, "Extract Here" on Linux, or "Extract all ..." on Windows.

To learn more about your shell, search Wikipedia for Windows Shell[37] or, for Mac/Linux, Bash Shell[38].

Installing Java

To compile and run the examples, you must first install the *Java development kit*. In this book we use JDK 8 (Java 1.8), although newer versions of Java are intended to be backward-compatible, so the examples should work with any version of Java as long as it's *at least* JDK 8. If you want to choose a newer version, I recommend a *Long-Term Support* version—at this writing, the most recent LTS version is JDK 17.

[37] https://en.wikipedia.org/wiki/Windows_shell
[38] https://en.wikipedia.org/wiki/Bash_(Unix_shell)

Windows

1. Follow the instructions at this link to Install Chocolatey[39].
2. At a shell prompt, type: `choco install jdk8`. This takes some time, but when it's finished Java is installed and the necessary environment variables are set.

Macintosh

The Mac comes with a much older version of Java that won't work for the examples in this book, so you must first update it to Java 8. You will need administration rights to perform these steps.

1. Follow the instructions at this link to Install HomeBrew[40]. Then at a shell prompt, execute `brew update` to make sure you have the latest changes.
2. At a shell prompt, execute `brew cask install java`.

Once HomeBrew and Java are installed, all other activities described in this book can be accomplished within a guest account, if that suits your needs.

Linux

Use the standard package installer with the following shell commands:

Ubuntu/Debian:

1. `sudo apt-get update`
2. `sudo apt-get install default-jdk`

Fedora/Redhat:

1. `su-c "yum install java-1.8.0-openjdk"`

Verify Your Installation

Open a new shell and type:

[39] https://chocolatey.org/
[40] http://brew.sh/

```
java -version
```

You should see something like the following (Version numbers and actual text will vary):

```
java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
```

If you see a message that the command is not found or not recognized, review the installation instructions in this chapter. If you still can't get it to work, check StackOverflow[41].

Installing and Running the Book Examples

Once you have Java installed, the process to install and run the book examples is the same for all platforms:

1. Download the book examples from the GitHub Repository[42].
2. `unzip` (as described in Basic Shell Operations) the downloaded file into the directory of your choice.
3. Use the Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux to browse to the directory where you unzipped `OnJava8-Examples`, and open a shell there.
4. If you're in the right directory, you should see files named `gradlew` and `gradlew.bat` in that directory, along with numerous other files and directories. The directories correspond to the chapters in the book.
5. At the shell prompt, type `gradlew run` (Windows) or `./gradlew run` (Mac/Linux).

The first time you do this, Gradle will install itself and numerous other packages, so it will take some time. After everything is installed, subsequent builds and runs are faster.

Note you must be connected to the Internet the first time you run `gradlew` so that Gradle can download the necessary packages.

[41] http://stackoverflow.com/search?q=installing+java
[42] https://github.com/BruceEckel/OnJava8-Examples/archive/master.zip

Basic Gradle Tasks

There are a large number of Gradle tasks automatically available with this book's build. Gradle uses an approach called *convention over configuration* which results in the availability of many tasks even if you're only trying to accomplish something very basic. Some of the tasks that "came along for the ride" with this book are inappropriate or don't successfully execute. Here is a list of the Gradle tasks you will typically use:

- `gradlew compileJava`: Compiles all the Java files in the book *that can be compiled* (some files don't compile, to demonstrate incorrect language usage).
- `gradlew run`: First compiles, then executes all the Java files in the book *that can be executed* (some files are library components).
- `gradlew test`: Executes all the *unit tests* (you'll learn about these in Validating Your Code).
- `gradlew` *chapter*:*ExampleName*: Compiles and runs a specific example program. For instance, `gradlew objects:HelloDate`.

Objects Everywhere

> "If we spoke a different language, we would perceive a somewhat different world."—*Wittgenstein*

Although it is based on C++, Java is more of a "pure" object-oriented language. Both C++ and Java are hybrid languages, but in Java the designers felt that the hybridization was not as important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backward compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features, which can make some aspects of C++ overly complicated.

The Java language assumes you're *only* writing object-oriented programs. Before you can begin you must shift your mindset into an object-oriented world. In this chapter you'll see the basic components of a Java program and learn that (almost) everything in Java is an object.

You Manipulate Objects with References

> "What's in a name? That which we call a rose, by any other word would smell as sweet."—*Shakespeare*, Romeo & Juliet

Every programming language manipulates elements in memory. Sometimes the programmer must be constantly aware of that manipulation. Do you manipulate the element directly, or use an indirect representation that requires special syntax (for example, pointers in C or C++)?

Java simplifies the issue by considering everything an object, using a single consistent syntax. Although you *treat* everything as an object, the identifier you manipulate

is actually a "reference" to an object.[43] You might imagine a television (the object) and a remote control (the reference). As long as you're holding this reference, you have a connection to the television, but when someone says, "Change the channel" or "Lower the volume," what you're manipulating is the reference, which in turn modifies the object. To move around the room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it. To hold a word or sentence, you create a `String` reference:

```
String s;
```

But here you've created *only* the reference, not an object. If you now decide to send a message to `s`, you get an error because `s` isn't actually attached to anything (there's no television). A safer practice is to always initialize a reference when you create it:

```
String s = "asdf";
```

This uses a special Java feature: Strings can be initialized with quoted text. You must use a more general type of initialization for other types of objects.

You Must Create All the Objects

The point of a reference is to connect it to an object. You usually create objects with the `new` operator. The keyword `new` says, "Make one of these." So in the preceding example, you can say:

[43]This can be a flashpoint. There are those who say, "Clearly, it's a pointer," but this presumes an underlying implementation. Also, the syntax of Java references is much more akin to C++ references than to pointers. In the 1st edition of *Thinking in Java*, I chose to invent a new term, "handle," because C++ references and Java references have some important differences. I was coming out of C++ and did not want to confuse the C++ programmers whom I assumed would be the largest audience for Java. In the 2nd edition of *Thinking in Java*, I decided that "reference" was the more commonly used term, and that anyone changing from C++ would have a lot more to cope with than the terminology of references, so they might as well jump in with both feet. However, there are people who disagree even with the term "reference." In one book I read that it was "completely wrong to say that Java supports pass by reference," because Java object identifiers (according to that author) are *actually* "object references." And (he goes on) everything is *actually* pass by value. So you're not passing by reference, you're "passing an object reference by value." One could argue for the precision of such convoluted explanations, but I think my approach simplifies the understanding of the concept without hurting anything (well, language lawyers may claim I'm lying to you, but I'll say that I'm providing an appropriate abstraction).

```
String s = new String("asdf");
```

Not only does this mean "Make a new `String`," but it also gives information about *how* to make the `String` by supplying an initial group of characters.

Java comes with a plethora of ready-made types in addition to `String`. On top of that, you can create your own types. In fact, creating new types is the fundamental activity in Java programming, and it's what you'll be learning about in the rest of this book.

Where Storage Lives

It's useful to visualize the way things are laid out while the program is running—in particular, how memory is arranged. There are five different places to store data:

1. **Registers**. This is the fastest storage because it exists in a place different from that of other storage: inside the *central processing unit* (CPU)[44]. However, the number of registers is severely limited, so registers are allocated as they are needed. You don't have direct control over register allocation, nor do you see any evidence in your programs that registers even exist (C & C++, on the other hand, allow you to suggest register allocation to the compiler).
2. **The stack**. This lives in the general random-access memory (RAM) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The Java system must know, while it is creating the program, the exact lifetime of all the items stored on the stack. This constraint places limits on the flexibility of your programs, so while some Java storage exists on the stack—in particular, object references—Java objects themselves are not placed on the stack.
3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all Java objects live. Unlike the stack, the compiler doesn't need to know how long objects must stay on the heap. Thus, there's a great deal of flexibility when using heap storage. Whenever you need an object, you write the code to create it using `new`, and the storage is allocated on the heap when that code is

[44]Most microprocessor chips do have additional *cache* memory but this is organized as traditional memory and not as registers.

executed. There's a price for this flexibility: It can take more time to allocate and clean up heap storage than stack storage (if you even *could* create objects on the stack in Java, as you can in C++). Over time, however, Java's heap allocation mechanism has become very fast, so this is not an issue for concern.

4. **Constant storage**. Constant values are often placed directly in the program code, which is safe because they can never change. Sometimes constants are cordoned off by themselves so they can be optionally placed in read-only memory (ROM), in embedded systems.[45]
5. **Non-RAM storage**. If data lives completely outside a program, it can exist while the program is not running, outside the control of the program. The two primary examples of this are *serialized objects*, where objects are turned into streams of bytes, generally sent to another machine, and *persistent objects*, where the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that exist on the other medium, and yet be resurrected into a regular RAM-based object when necessary. Java provides support for *lightweight persistence*. Libraries such as JDBC[46] and Hibernate[47] provide more sophisticated support for storing and retrieving object information using databases.

Special Case: Primitive Types

One group of types that you'll often use gets special treatment. You can think of these as "primitive" types. The reason for the special treatment is that creating an object with `new`—especially a small, simple variable—isn't very efficient, because `new` places objects on the heap. For these types Java falls back on the approach taken by C and C++. That is, instead of creating the variable using `new`, an "automatic" variable is created that is *not a reference*. The variable holds the value directly, and it's placed on the stack, so it's much more efficient.

Java specifies the size of each primitive type, and these sizes don't change from one machine architecture to another as they do in some languages. This size invariance is one reason Java programs are more portable than programs in some other languages.

[45] An example of this is the `String` pool. All literal `Strings` and `String`-valued constant expressions are interned automatically and put into special static storage.

[46] http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[47] http://hibernate.org/

Primitive	Size	Min	Max	Wrapper
`boolean`	—	—	—	`Boolean`
`char`	16 bits	Unicode 0 `\u0000`	65,535 `\uffff`	`Character`
`byte`	8 bits	-128	+127	`Byte`
`short`	16 bits	-32,768	+32,767	`Short`
`int`	32 bits	-2^{31}	$+2^{31}-1$	`Integer`
`long`	64 bits	-2^{63}	$+2^{63}-1$	`Long`
`float`	32 bits	IEEE754	IEEE754	`Float`
`double`	64 bits	IEEE754	IEEE754	`Double`
`void`	—	—	—	`Void`

All numeric types are signed, so don't look for unsigned types.

The size of the `boolean` type is not explicitly specified; it is defined to take the literal values `true` or `false`.

"Wrapper" classes for primitive data types create a non-primitive object on the heap to represent that primitive type. For example:

```
char c = 'x';
Character ch = new Character(c);
```

You can also use:

```
Character ch = new Character('x');
```

Autoboxing automatically converts a primitive to a wrapped object:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

The reasons for wrapping primitives are shown in a later chapter.

High-Precision Numbers

Java includes two classes for performing high-precision arithmetic: `BigInteger` and `BigDecimal`. Although these fit approximately the same category as the "wrapper" classes, neither one has a corresponding primitive.

Both classes have methods that provide analogues for the operations you perform on primitive types. That is, you can do anything with a `BigInteger` or `BigDecimal` you can with an `int` or `float`, it's just that you must use method calls instead of operators. Also, because there are more calculations involved, the operations are slower. You're exchanging speed for accuracy.

`BigInteger` supports arbitrary-precision integers. This means you can accurately represent integral values of any size without losing any information during operations.

`BigDecimal` is for arbitrary-precision fixed-point numbers; you can use these for accurate monetary calculations, for example.

Consult the JDK documentation for details about these two classes.

Arrays in Java

Many programming languages support some kind of array. Using arrays in C and C++ is perilous because those arrays are only blocks of memory. If a program accesses the array outside of its memory block or uses the memory before initialization (common programming errors), the results are unpredictable.

One of the primary goals of Java is safety, so many of the problems that plague programmers in C and C++ are not repeated in Java. A Java array is guaranteed to be initialized and cannot be accessed outside of its range. This range checking comes at the price of having a small amount of memory overhead for each array as well as extra time to verify the index at runtime, but the assumption is that the safety and increased productivity are worth the expense (and Java can often optimize these operations).

When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: `null`. When Java sees `null`, it recognizes that the reference in question isn't pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that's still `null`, the problem is reported at runtime. Thus, typical array errors are prevented in Java.

You can also create an array of primitives. The compiler guarantees initialization by zeroing the memory for that array.

Arrays are covered in detail later in the book, and specifically in the Arrays chapter.

Comments

There are two types of comments in Java. The first are traditional C-style comments which begin with a `/*` and continue, possibly across many lines, until a `*/`. Note that many programmers begin each line of a multiline comment with a `*`, so you'll often see:

```
/* This is a comment
 * that continues
 * across lines
 */
```

Remember, however, that everything inside the `/*` and `*/` is ignored, so there's no difference if you say instead:

```
/* This is a comment that
continues across lines */
```

The second form of comment comes from C++. It is the single-line comment, which starts with a `//` and continues until the end of the line. This type of comment is convenient and commonly used because it's easy. So you often see:

```
// This is a one-line comment
```

You Never Need to Destroy an Object

In some programming languages, managing storage lifetime requires significant effort. How long does a variable last? If you are supposed to destroy it, when should you? Confusion over storage lifetime can lead to many bugs, and this section shows how Java simplifies the issue by releasing storage for you.

Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and Java, scope is determined by the placement of curly braces {}. Here is a fragment of Java code demonstrating scope:

```
{
  int x = 12;
  // Only x available
  {
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q is "out of scope"
}
```

A variable defined within a scope is available only until the end of that scope.

Indentation makes Java code easier to read. Because Java is a free-form language, the extra spaces, tabs, and carriage returns do not affect the resulting program.

You *cannot* do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
    int x = 96; // Illegal
  }
}
```

The Java compiler will announce that the variable `x` has already been defined. Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed, because the Java designers thought it led to confusing programs.

Scope of Objects

Java objects do not have the same lifetimes as primitives. When you create a Java object using `new`, it persists past the end of the scope. Thus, if you say:

```
{
  String s = new String("a string");
} // End of scope
```

the reference `s` vanishes at the end of the scope. However, the `String` object that `s` points to is still occupying memory. In this bit of code, there is no way to access the object after the end of the scope, because the only reference to it is out of scope. In later chapters you'll see how the reference to the object can be passed around and duplicated during the course of a program.

Because objects created with `new` exist as long as you need them, a whole slew of C++ programming problems vanish in Java. In C++ you must not only make sure that the objects stay around as long as necessary, you must also destroy the objects when you're done with them.

This brings up a question. If Java leaves the objects lying around, what keeps them from filling up memory and halting your program, which is exactly the kind of problem that occurs in C++? In Java, a bit of magic happens: the *garbage collector* looks at all the objects created with `new` and finds those that are no longer referenced. It then releases the memory for those objects, so the memory can be used for new objects. This means you don't worry about reclaiming memory yourself. You simply create objects, and when you no longer need them, they go away by themselves. This prevents an important class of programming problem: the so-called "memory leak," when a programmer forgets to release memory.

Creating New Data Types: `class`

If everything is an object, what determines how a particular class of object looks and behaves? Put another way, what establishes the *type* of an object? You might expect a keyword called "type," and that would certainly make sense. Historically, however, most object-oriented languages use the keyword `class` to describe a new kind of object. The `class` keyword (so common it will often not be bold-faced throughout this book) is followed by the name of the new type. For example:

```
class ATypeName {
  // Class body goes here
}
```

This introduces a new type, although here the class body consists only of a comment, so there is not too much you can do with it. However, you can create an object of `ATypeName` using `new`:

```
ATypeName a = new ATypeName();
```

You can't tell it to do much of anything—that is, you cannot send it any interesting messages—until you define some methods for it.

Fields

When you define a class, you can put two types of elements in your class: *fields* (sometimes called *data members*), and *methods* (sometimes called *member functions*). A field is an object of any type you can talk to via its reference. A field can also be a primitive type. If it is a reference to an object, you must initialize that reference to connect it to an actual object (using `new`, as seen earlier).

Each object keeps its own storage for its fields. Ordinarily, fields are not shared among objects. Here is an example of a class with some fields:

```
class DataOnly {
  int i;
  double d;
  boolean b;
}
```

This class doesn't do anything except hold data. As before, you create an object like this:

```
DataOnly data = new DataOnly();
```

You can assign values to the fields by referring to object members. To do this, you state the name of the object reference, followed by a period (dot), followed by the name of the member inside the object:

```
objectReference.member
```

For example:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

What if your object contains other objects that contain data you want to modify? You just keep "connecting the dots." For example:

```
myPlane.leftTank.capacity = 100;
```

You can nest many objects this way (although such a design might become confusing).

Default Values for Primitive Members

When a primitive data type is a field in a class, it is guaranteed to get a default value if you do not initialize it:

Primitive	Default
`boolean`	`false`
`char`	`\u0000` (`null`)
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`

The default values are only what Java guarantees when the variable is used *as a member of a class*. This ensures that primitive fields will always be initialized (something C++ doesn't do), reducing a source of bugs. However, this initial value might not be correct or even legal for the program you are writing. It's best to always explicitly initialize your variables.

This guarantee doesn't apply to *local variables*—those that are not fields of a class. Thus, if within a method definition you have:

```
int x;
```

Then `x` will get some arbitrary value (as it does in C and C++); it will not automatically be initialized to zero. You are responsible for assigning an appropriate value before you use `x`. If you forget, Java definitely improves on C++: You get a compile-time error telling you the variable might not be initialized. (C++ compilers often warn you about uninitialized variables, but in Java these are errors.)

Methods, Arguments, and Return Values

In many languages (like C and C++), the term *function* is used to describe a named subroutine. In Java, we use the term *method*, as in "a way to do something."

Methods in Java determine the messages an object can receive. The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
ReturnType methodName( /* Argument list */ ) {
  // Method body
}
```

`ReturnType` indicates the type of value produced by the method when you call it. The argument list gives the types and names for the information you pass into the method. The method name and argument list are collectively called the *signature* of the method. The signature uniquely identifies that method.

Methods in Java can only be created as part of a class. A method can be called only for an object[48], and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile time.

You call a method for an object by giving the object reference followed by a period (dot), followed by the name of the method and its argument list, like this:

```
objectReference.methodName(arg1, arg2, arg3);
```

Consider a method `f()` that takes no arguments and returns a value of type `int`. For a reference `a` that accepts calls to `f()`, you can say this:

```
int x = a.f();
```

The type of the return value must be compatible with the type of `x`.

This act of calling a method is sometimes termed *sending a message to an object*. In the preceding example, the message is `f()` and the object is `a`. Object-oriented programming can be summarized as "sending messages to objects."

The Argument List

The method argument list specifies the information you pass into the method. As you might guess, this information—like everything else in Java—takes the form of objects. The argument list must specify the object types and the name of each

[48] `static` methods, which you'll learn about soon, can be called *for the class*, without an object.

object. As always, where you seem to be handing objects around, you are actually passing references.[49] The type of the reference must be correct, however. If a `String` argument is expected, you must pass in a `String` or the compiler will give an error.

Here is the definition for a method that takes a `String` as its argument. It must be placed within a class for it to compile:

```
int storage(String s) {
  return s.length() * 2;
}
```

This method calculates and delivers the number of bytes required to hold the information in a particular `String`. The argument `s` is of type `String`. Once `s` is passed into `storage()`, you can treat it like any other object—you can send it messages. Here, we call `length()`, which is a `String` method that returns the number of characters in a `String`. The size of each `char` in a `String` is 16 bits, or two bytes.

You can also see the `return` keyword, which does two things. First, it means "Leave the method, I'm done." Second, if the method produces a value, that value is placed right after the `return` statement. Here, the return value is produced by evaluating the expression `s.length() * 2`.

You can return any type you want, but if you don't return anything at all, you do so by indicating that the method produces `void` (nothing). Here are some examples:

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

When the return type is `void`, the `return` keyword is used only to exit the method, and is therefore unnecessary if called at the end of the method. You can return from a method at any point, but if you've given a non-`void` return type, the compiler will force you to return the appropriate type of value regardless of where you return.

It might look like a program is just a bunch of objects with methods that take other objects as arguments and send messages to those other objects. That is indeed much

[49] With the usual exception of the aforementioned "special" data types `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In general, though, you pass objects, which really means you pass references to objects.

of what goes on, but in the following Operators chapter you'll learn how to do the detailed low-level work by making decisions within a method. For this chapter, sending messages will suffice.

Writing a Java Program

There are several other issues you must understand before seeing your first Java program.

Name Visibility

A problem in any programming language is the control of names. If you use a name in one module of the program, and another programmer uses the same name in another module, how do you distinguish one name from another and prevent the two names from "clashing?" In C this is especially challenging because a program is often an unmanageable sea of names. C++ classes (on which Java classes are modeled) nest functions within classes so they cannot clash with function names nested within other classes. However, C++ continues to allow global data and global functions, so clashing is still possible. To solve this problem, C++ introduced *namespaces* using additional keywords.

Java avoided all these problems by taking a fresh approach. To produce an unambiguous name for a library, the Java creators want you to use your Internet domain name in reverse, because domain names are guaranteed to be unique. My domain name is `MindviewInc.com` so my `foibles` utility library is named `com.mindviewinc.utility.foibles`. Following your reversed domain name, the dots are intended to represent subdirectories.

In Java 1.0 and Java 1.1 the domain extensions `com`, `edu`, `org`, `net`, etc., were capitalized by convention, so the library was `Com.mindviewinc.utility.foibles`. Partway through the development of Java 2, however, they discovered this caused problems, so now the entire package name is lowercase.

This mechanism means all your files automatically live in their own namespaces, and each class within a file has a unique identifier. This way, the language prevents name clashes.

Using reversed URLs was a new approach to namespaces, never before tried in another language. Java has a number of these "inventive" approaches to problems. As you might imagine, adding a feature without experimenting with it first risks discovering problems with that feature in the future, after the feature is used in production code, typically when it's too late to do anything about it (some mistakes were bad enough to actually remove things from the language).

The problem with associating namespaces with file paths using reversed URLs is by no means one that causes bugs, but it does make it challenging to manage source code. By using `com.mindviewinc.utility.foibles`, I create a directory hierarchy with "empty" directories `com` and `mindviewinc` whose only job is to reflect the reversed URL. This approach seemed to open the door to what you will encounter in production Java programs: deep directory hierarchies filled with empty directories, not just for the reversed URLs but also to capture other information. These long paths are basically being used to store data about what is in the directory. If you expect to use directories in the way they were originally designed, this approach lands anywhere from "frustrating" to "maddening," and for production Java code you are essentially forced to use one of the IDEs specifically designed to manage code that is laid out in this fashion, such as IntelliJ IDEA. Indeed, IDEs often manage and create the deep empty directory hierarchies for you.

For this book's examples, I didn't want to burden you with the extra annoyance of the deep hierarchies, which would have effectively required you to learn one of the big IDEs before getting started. The examples for each chapter are in a shallow subdirectory with a name reflecting the chapter title. This caused me occasional struggles with tools that follow the deep-hierarchy approach.

Using Other Components

Whenever you use a predefined class in your program, the compiler must locate that class. In the simplest case, the class already exists in the source-code file it's being called from. In that case, you simply use the class—even if the class doesn't get defined until later in the file (Java eliminates the so-called "forward referencing" problem).

What about a class that exists in some other file? You might think the compiler should be smart enough to go and find it, but there is a problem. Imagine you use a class with a particular name, but more than one definition for that class exists (presumably

these are different definitions). Or worse, imagine that you're writing a program, and as you're building it you add a new class to your library that conflicts with the name of an existing class.

To solve this problem, you must eliminate all potential ambiguities by telling the Java compiler exactly what classes you want using the `import` keyword. `import` tells the compiler to bring in a package, which is a library of classes. (In other languages, a library could consist of functions and data as well as classes, but remember that all activities in Java take place within classes.)

Much of the time you'll use components from the standard Java libraries that come with your compiler. With these, you don't worry about long, reversed domain names; you just say, for example:

```
import java.util.ArrayList;
```

This tells the compiler to use Java's `ArrayList` class, located in its `util` library.

However, `util` contains several classes, and you might want to use several of them without declaring them all explicitly. This is easily accomplished by using `*` to indicate a wild card:

```
import java.util.*;
```

The examples in this book are small and for simplicity's sake we'll usually use the `*` form. However, many style guides specify that each class should be individually imported.

The `static` Keyword

Creating a class describes the look of its objects and the way they behave. You don't actually get an object until you create one using `new`, at which point storage is allocated and methods become available.

This approach is insufficient in two cases. Sometimes you want only a single, shared piece of storage for a particular field, regardless of how many objects of that class are created, or even if no objects are created. The second case is if you need a method that isn't associated with any particular object of this class. That is, you need a method you can call even if no objects are created.

The `static` keyword (adopted from C++) produces both these effects. When you say something is `static`, it means the field or method is not tied to any particular object instance. Even if you've never created an object of that class, you can call a `static` method or access a `static` field. With ordinary, non-`static` fields and methods, you must create an object and use that object to access the field or method, because non-`static` fields and methods must target a particular object.[50]

Some object-oriented languages use the terms *class data* and *class methods*, meaning that the data and methods exist only for the class as a whole, and not for any particular objects of the class. Sometimes Java literature uses these terms too.

To make a field or method `static`, you place the keyword before the definition. The following produces and initializes a `static` field:

```
class StaticTest {
  static int i = 47;
}
```

Now even if you make two `StaticTest` objects, there is still only one piece of storage for `StaticTest.i`. Both objects share the same `i`. For example:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

Both `st1.i` and `st2.i` have the same value of 47 because they are the same piece of memory.

There are two ways to refer to a `static` variable. As in the preceding example, you can name it via an object; for example, `st2.i`. You can also refer to it directly through its class name, something you cannot do with a non-`static` member:

```
StaticTest.i++;
```

The ++ operator adds one to the variable. Now both `st1.i` and `st2.i` have the value 48.

Using the class name is the preferred way to refer to a `static` variable because it emphasizes the variable's `static` nature[51].

[50] `static` methods don't require objects to be created before they are used, so they cannot *directly* access non-`static` members or methods by calling those other members without referring to a named object (because non-`static` members and methods must be tied to a particular object).

[51] In some cases it also gives the compiler better opportunities for optimization.

Similar logic applies to `static` methods. You can refer to a `static` method either through an object as you can with any method, or with the special additional syntax `ClassName.method()`. You define a `static` method like this:

```
class Incrementable {
  static void increment() { StaticTest.i++; }
}
```

The `Incrementable` method `increment()` increments the `static int i` using the `++` operator. You can call `increment()` in the typical way, through an object:

```
Incrementable sf = new Incrementable();
sf.increment();
```

However, the preferred approach is to call it directly through its class:

```
Incrementable.increment();
```

`static` applied to a field definitely changes the way the data is created—one for each class versus the non-`static` one for each object. When applied to a method, `static` allows you to call that method without creating an object. This is essential, as you will see, in defining the `main()` method that is the entry point for running an application.

Your First Java Program

Finally, here's the first complete program. It starts by displaying a `String`, followed by the date, using the `Date` class from the Java standard library.

```
// objects/HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

In this book I treat the first line specially; it's always a comment line containing the path information to the file (using the directory name `objects` for this chapter) followed by the file name. I have tools to automatically extract and test the book's code based on this information, and you will easily find the code example in the repository[52] by referring to the first line.

At the beginning of each program file, you must place `import` statements to bring in any extra classes you need for the code in that file. I say "extra" because there's a certain library of classes automatically included in every Java file: `java.lang`. Start up your Web browser and look at the documentation from Oracle. If you haven't downloaded the JDK documentation from the Oracle Java site, do so now[53], or find it on the Internet. If you look at the list of packages, you'll see all the different class libraries that come with Java. Select `java.lang`. This will bring up a list of all the classes that are part of that library. Because `java.lang` is implicitly included in every Java code file, these classes are automatically available. There's no `Date` class listed in `java.lang`, which means you must import another library to use that. If you don't know the library where a particular class is, or if you want to see all classes, select "Tree" in the Java documentation. Now you can find every single class that comes with Java. Use the browser's "find" function to find `Date`. You'll see it listed as `java.util.Date`, which tells you it's in the `util` library and that you must `import java.util.*` to use `Date`.

If inside the documentation you select `java.lang`, then `System`, you'll see that the `System` class has several fields, and if you select `out`, you'll discover it's a `static PrintStream` object. It's `static` so you don't need to use `new`—the `out` object is always there, and you can just use it. What you can do with this `out` object is

[52] https://github.com/BruceEckel/OnJava8-Examples

[53] java.oracle.com. Note the documentation doesn't come packed with the JDK; you must do a separate download to get it.

determined by its type: `PrintStream`. Conveniently, `PrintStream` is shown in the description as a hyperlink, so if you click on that, you'll see a list of all the methods you can call for `PrintStream`. There are quite a few, and these are covered later in the book. For now all we're interested in is `println()`, which in effect means "Print what I'm giving you out to the console and end with a newline." Thus, in any Java program you can write something like this:

```
System.out.println("A String of things");
```

whenever you want to display information to the console.

One of the classes in the file must have the same name as the file. (The compiler complains if you don't do this.) When you're creating a standalone program such as this one, the class with the name of the file must contain an *entry point* from which the program starts. This special method is called `main()`, with the following signature and return type:

```
public static void main(String[] args) {
```

The `public` keyword means the method is available to the outside world (described in detail in the Implementation Hiding chapter). The argument to `main()` is an array of `String` objects. The `args` won't be used in the current program, but the Java compiler insists they be there because they hold the arguments from the command line.

The line that prints the date is interesting:

```
System.out.println(new Date());
```

The argument is a `Date` object that is only created to send its value (automatically converted to a `String`) to `println()`. As soon as this statement is finished, that `Date` is unnecessary, and the garbage collector can come along and get it anytime. We don't worry about cleaning it up.

When you look at the JDK documentation, you see that `System` has many other useful methods (one of Java's assets is its large set of standard libraries). For example:

```
// objects/ShowProperties.java

public class ShowProperties {
  public static void main(String[] args) {
    System.getProperties().list(System.out);
    System.out.println(System.getProperty("user.name"));
    System.out.println(
      System.getProperty("java.library.path"));
  }
}
/* Output: (First 20 Lines)
-- listing properties --
java.runtime.name=OpenJDK Runtime Environment
sun.boot.library.path=C:\Program Files\OpenJDK\java-
se-8u41...
java.vm.version=25.40-b25
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
java.vm.name=OpenJDK Client VM
file.encoding.pkg=sun.io
user.script=
user.country=US
sun.java.launcher=SUN_STANDARD
sun.os.patch.level=
java.vm.specification.name=Java Virtual Machine
Specification
user.dir=C:\Git\OnJava8\ExtractedExamples\objects
java.runtime.version=1.8.0_41-b04
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\Program Files\OpenJDK\java-
se-8u41...
os.arch=x86
java.io.tmpdir=C:\Users\Bruce\AppData\Local\Temp\
                  ...
*/
```

The first line in `main()` displays all "properties" from the system where you are running the program, so it gives you environment information. The `list()` method sends the results to its argument, `System.out`. You will see later in the book that you can send the results elsewhere, to a file, for example. You can also ask for a specific property—here, `user.name` and `java.library.path`.

The `/* Output:` tag at the end of the file indicates the beginning of the output generated by this file. Most examples in this book that produce output will contain the output in this commented form, so you see the output and know it is correct. The tag allows the output to be automatically updated into the text of this book after being checked with a compiler and executed.

Compiling and Running

To compile and run this program, and all other programs in this book, you must first have a Java programming environment. The installation process is described in the Book Examples[54]. Following these instructions will install the Java Developer's Kit (JDK).

After you've installed the JDK, move to the subdirectory named `objects` and type:

```
javac HelloDate.java
```

This command should produce no response. If you get any kind of an error message, it means you haven't installed the JDK properly and you must investigate those problems.

On the other hand, if you just get your command prompt back, you can type:

```
java HelloDate
```

and you'll see the message and the date as output.

This is the process to compile and run each program (containing a `main()`) in this book[55]. However, the source code for this book also has a file called `build.gradle` in the root directory, containing the *Gradle* configuration for automatically building, testing, and running the files for the book. When you run the `gradlew` command for the first time, Gradle will automatically install itself (assuming you have Java installed).

[54] https://www.onjava8.com/examples/

[55] For every program in this book to compile and run through the command line, you might also need to set your CLASSPATH.

Coding Style

The style described in the document *Code Conventions for the Java Programming Language*[56] is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don't use underscores to separate the names), and the first letter of each embedded word is capitalized, such as:

```
class AllTheColorsOfTheRainbow { // ...
```

This is sometimes called "camel-casing." For almost everything else—methods, fields (member variables), and object reference names—the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase. For example:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

The user must also type these long names, so be merciful.

The Java code you find in the Oracle libraries also follows the placement of open-and-close curly braces in this book.

Summary

This chapter shows you just enough Java so you understand how to write a simple program. You've also seen an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "Do this, then do that, then do something else." The next two chapters will introduce the basic operators used in Java programming, and show you how to control the flow of your program.

[56](Search the Internet; also look for "Google Java Style"). To keep code listings narrow for this book, not all these guidelines could be followed, but you'll see that the style I use here matches the Java standard as much as possible.

Operators

> Operators manipulate data.

Because Java was inherited from C++, most of its operators are familiar to C and C++ programmers. Java also adds some improvements and simplifications.

If you already know C or C++ syntax, you can skim through this chapter and the next, looking for places where Java is different from those languages.

Using Java Operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. Addition and unary plus (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language.

All operators produce a value from their operands. In addition, some operators change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but keep in mind that the value produced is available for your use, just as in operators without side effects.

Almost all operators work only with primitives. The exceptions are =, == and !=, which work with all objects (and are a point of confusion for objects). In addition, the `String` class supports + and +=.

Precedence

Operator precedence defines expression evaluation when several operators are present. Java has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, and use parentheses to make the order of evaluation explicit. For example, look at statements **[1]** and **[2]**:

```
// operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;          // [1]
    int b = x + (y - 2)/(2 + z);      // [2]
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
}
/* Output:
a = 5
b = 1
*/
```

These statements look roughly the same, but from the output you see they have very different meanings depending on the use of parentheses.

Notice that `System.out.println()` uses the `+` operator. In this context, `+` means "`String` concatenation" and, if necessary, "`String` conversion." When the compiler sees a `String` followed by a `+` followed by a non-`String`, it attempts to convert the non-`String` into a `String`. The output shows it successfully converts from `int` into `String` for `a` and `b`.

Assignment

The operator = performs assignment. It means "Take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)." An rvalue is any constant, variable, or expression that produces a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store the value.) For instance, you can assign a constant value to a variable:

```
a = 4;
```

but you cannot assign anything to a constant value—it cannot be an lvalue. (You can't say `4 = a;`.)

Assigning primitives is straightforward. The primitive holds the actual value and not a reference to an object, so when you assign primitives, you copy the contents from

one place to another. For example, if you say `a = b` for primitives, the contents of `b` are copied into `a`. If you then go on to modify `a`, `b` is naturally unaffected by this modification. As a programmer, this is what you can expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another," you're actually copying a reference from one place to another. This means if you say `c = d` for objects, you end up with both `c` and `d` pointing to the object where, originally, only `d` pointed. This example demonstrates the behavior:

```
// operators/Assignment.java
// Assignment with objects is a bit tricky

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    System.out.println("1: t1.level: " + t1.level +
      ", t2.level: " + t2.level);
    t1 = t2;
    System.out.println("2: t1.level: " + t1.level +
      ", t2.level: " + t2.level);
    t1.level = 27;
    System.out.println("3: t1.level: " + t1.level +
      ", t2.level: " + t2.level);
  }
}
/* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*/
```

The `Tank` class is simple, and two instances (`t1` and `t2`) are created within `main()`. The `level` field within each `Tank` is given a different value, then `t2` is assigned to

`t1`, and `t1` is changed. In many programming languages you expect `t1` and `t2` to be independent at all times, but because you've assigned a reference, changing the `t1` object appears to change the `t2` object as well! This is because both `t1` and `t2` contain references that point to the same object. (The original reference that was in `t1`, that pointed to the object holding a value of 9, was overwritten during the assignment and effectively lost; its object is cleaned up by the garbage collector.)

This phenomenon is often called *aliasing*, and it's a fundamental way that Java works with objects. But what if you don't want aliasing to occur here? You can forego the assignment and say:

```
t1.level = t2.level;
```

This retains the two separate objects instead of discarding one and tying `t1` and `t2` to the same object. Manipulating the fields within objects goes against Java design principles. This is a nontrivial topic, so keep in mind that assignment for objects can add surprises.

Aliasing During Method Calls

Aliasing will also occur when you pass an object into a method:

```
// operators/PassObject.java
// Passing objects to methods might not be
// what you're used to

class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.println("2: x.c: " + x.c);
```

```
  }
}
/* Output:
1: x.c: a
2: x.c: z
*/
```

In many programming languages, the method `f()` appears to make a copy of its argument `Letter y` inside the scope of the method. But once again a reference is passed, so the line

```
y.c = 'z';
```

is actually changing the object *outside* of `f()`.

Aliasing and its solution is a complex issue covered in the Appendix: Passing and Returning Objects. You're aware of it now so you can watch for pitfalls.

Mathematical Operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (+), subtraction (-), division (/), multiplication (*) and modulus (%, which produces the remainder from division). Integer division truncates, rather than rounds, the result.

Java also uses the shorthand notation from C/C++ that performs an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable `x` and assign the result to `x`, use: `x += 4`.

This example shows the mathematical operators:

```
// operators/MathOps.java
// The mathematical operators
import java.util.*;

public class MathOps {
  public static void main(String[] args) {
    // Create a seeded random number generator:
    Random rand = new Random(47);
    int i, j, k;
    // Choose value from 1 to 100:
    j = rand.nextInt(100) + 1;
    System.out.println("j : " + j);
    k = rand.nextInt(100) + 1;
    System.out.println("k : " + k);
    i = j + k;
    System.out.println("j + k : " + i);
    i = j - k;
    System.out.println("j - k : " + i);
    i = k / j;
    System.out.println("k / j : " + i);
    i = k * j;
    System.out.println("k * j : " + i);
    i = k % j;
    System.out.println("k % j : " + i);
    j %= k;
    System.out.println("j %= k : " + j);
    // Floating-point number tests:
    float u, v, w; // Applies to doubles, too
    v = rand.nextFloat();
    System.out.println("v : " + v);
    w = rand.nextFloat();
    System.out.println("w : " + w);
    u = v + w;
    System.out.println("v + w : " + u);
    u = v - w;
    System.out.println("v - w : " + u);
    u = v * w;
    System.out.println("v * w : " + u);
    u = v / w;
    System.out.println("v / w : " + u);
    // The following also works for char,
    // byte, short, int, long, and double:
```

```
    u += v;
    System.out.println("u += v : " + u);
    u -= v;
    System.out.println("u -= v : " + u);
    u *= v;
    System.out.println("u *= v : " + u);
    u /= v;
    System.out.println("u /= v : " + u);
  }
}
/* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*/
```

To generate numbers, the program first creates a `Random` object. If you create a `Random` object with no arguments, Java uses the current time as a seed for the random number generator, and will thus produce different output for each execution of the program. However, in the examples in this book, it is important that the output at the end of each example be as consistent as possible so it can be verified with external tools. By providing a *seed* (an initialization value for the random number generator that always produces the same sequence for a particular seed value) when creating the `Random` object, the same random numbers are generated each time the program

is executed, so the output is verifiable.[57] To generate more varying output, feel free to remove the seed in the examples in the book.

The program generates several different types of random numbers with the `Random` object by calling the methods `nextInt()` and `nextFloat()` (you can also call `nextLong()` or `nextDouble()`). The argument to `nextInt()` sets the upper bound on the generated number. The lower bound is zero, which we don't want because of the possibility of a divide-by-zero, so the result is offset by one.

Unary Minus and Plus Operators

The unary minus (-) and unary plus (+) are the same operators as binary minus and plus. The compiler figures out which use is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler is able to figure out:

```
x = a * -b;
```

but the reader might get confused, so it is sometimes clearer to say:

```
x = a * (-b);
```

Unary minus inverts the sign on the data. Unary plus provides symmetry with unary minus, but its only effect is to promote smaller-type operands to `int`.

Auto Increment and Decrement

Java, like C, has a number of shortcuts. Shortcuts can make code much easier to type, and either easier or harder to read.

Two of the nicer shortcuts are the increment and decrement operators (often called the auto-increment and auto-decrement operators). The decrement operator is - - and

[57] As an undergraduate, I attended Pomona College for two years, where the number 47 was considered a "magic number." See the Wikipedia article.

means "decrease by one unit." The increment operator is ++ and means "increase by one unit." If a is an int, for example, the expression ++a is equivalent to a = a + 1. Increment and decrement operators not only modify the variable, but also produce the value of the variable as a result.

There are two versions of each type of operator, often called *prefix* and *postfix*. *Pre-increment* means the ++ operator appears before the variable, and *post-increment* means the ++ operator appears after the variable. Similarly, *pre-decrement* means the --operator appears before the variable, and *post-decrement* means the -- operator appears after the variable. For pre-increment and pre-decrement (i.e., ++a or --a), the operation is performed and the value is produced. For post-increment and post-decrement (i.e., a++ or a--), the value is produced, then the operation is performed.

```
// operators/AutoInc.java
// The ++ and -- operators

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    System.out.println("i: " + i);
    System.out.println("++i: " + ++i); // Pre-increment
    System.out.println("i++: " + i++); // Post-increment
    System.out.println("i: " + i);
    System.out.println("--i: " + --i); // Pre-decrement
    System.out.println("i--: " + i--); // Post-decrement
    System.out.println("i: " + i);
  }
}
/* Output:
i: 1
++i: 2
i++: 2
i: 3
--i: 2
i--: 2
i: 1
*/
```

For the prefix form, you get the value after the operation is performed, but with the postfix form, you get the value before the operation is performed. These are the only

operators, other than those involving assignment, that have side effects—they change the operand rather than just using its value.

The increment operator is one explanation for the name C++, implying "one step beyond C." In an early Java speech, Bill Joy (one of the Java creators), said that "`Java = C++--`" (C plus plus minus minus), suggesting that Java is C++ with the unnecessary hard parts removed, and therefore a much simpler language. As you progress, you'll see that many parts are simpler, and yet in other ways Java isn't much easier than C++.

Relational Operators

Relational operators produce a `boolean` result indicating the relationship between the values of the operands. A relational expression produces `true` if the relationship is true, and `false` if the relationship is untrue. The relational operators are less than (`<`), greater than (`>`), less than or equal to (`<=`), greater than or equal to (`>=`), equivalent (`==`) and not equivalent (`!=`). Equivalence and non-equivalence work with all primitives, but the other comparisons won't work with type `boolean`. Because `boolean` values can only be `true` or `false`, "greater than" or "less than" doesn't make sense.

Testing Object Equivalence

The relational operators == and != work with all objects, but their results can be confusing:

```
// operators/Equivalence.java

public class Equivalence {
  static void show(String desc, Integer n1, Integer n2) {
    System.out.println(desc + ":");
    System.out.printf(
      "%d==%d %b %b%n", n1, n2, n1 == n2, n1.equals(n2));
  }
  @SuppressWarnings("deprecation")
  public static void test(int value) {
    Integer i1 = value;                              // [1]
```

```
    Integer i2 = value;
    show("Automatic", i1, i2);
    // Old way, deprecated since Java 9:
    Integer r1 = new Integer(value);                // [2]
    Integer r2 = new Integer(value);
    show("new Integer()", r1, r2);
    // Preferred since Java 9:
    Integer v1 = Integer.valueOf(value);            // [3]
    Integer v2 = Integer.valueOf(value);
    show("Integer.valueOf()", v1, v2);
    // Primitives can't use equals():
    int x = value;                                  // [4]
    int y = value;
    // x.equals(y); // Doesn't compile
    System.out.println("Primitive int:");
    System.out.printf("%d==%d %b%n", x, y, x == y);
  }
  public static void main(String[] args) {
    test(127);
    test(128);
  }
}
/* Output:
Automatic:
127==127 true true
new Integer():
127==127 false true
Integer.valueOf():
127==127 true true
Primitive int:
127==127 true
Automatic:
128==128 false true
new Integer():
128==128 false true
Integer.valueOf():
128==128 false true
Primitive int:
128==128 true
*/
```

`show()` compares the behavior of == versus the method `equals()` that exists for

all objects—in this case, `Integer` objects. The `printf()` format argument uses the specifier `%d` for `int` output, `%b` for Boolean output, and `%n` to produce a newline.

(For "not equals" comparisons, use `n1 != n2` and `!n1.equals(n2)`).

In `test()`, integer values are created in four different ways:

- **[1]**: Automatic conversion to `Integer`. These are translated into calls to `Integer.valueOf()`.
- **[2]**: Using standard `new` object-creation syntax. Originally this was the preferred way to create "wrapped/boxed" `Integer` objects.
- **[3]**: Starting with Java 9, `valueOf()` is preferred over **[2]**. If you try to use **[2]** with Java 9, you'll get a warning and a suggestion to use **[3]** instead. It is difficult to determine whether **[3]** is preferred over **[1]**, and **[1]** seems much cleaner.
- **[4]**: As primitive `int`s. Notice that these are *not* passed to `show()` and you cannot call `equals()` for primitives; you can only use `==` and `!=`. This is seen in the use of `==` in the `printf()` call.

The `@SuppressWarnings("deprecation")` is not necessary for Java 8, but is included in case you compile the code with Java 9 or newer.

For a `value` of 127, the comparisons produce the results you expect, *except* for **[2]** which produces `false` for `==`. Although the *contents* of the objects are the same, the references point to different objects in memory. The operators `==` and `!=` compare object references, and have different behavior *depending on how the `Integer` objects are created*—presumably, **[1]** and **[3]** yield `Integer`s that point to the same storage in memory. `Integer` values from -128 through 127 produce this behavior for `==` and `!=`, but outside that range they do not, as seen with `test(128)`.

If you're using `Integer` you must only use `equals()`. If you accidentally use `==` and `!=` for `Integer` and don't test it for values outside -128 through 127, your code will pass but will be quietly broken. If you're using primitive `int`s then you *can't* use `equals()` and *must* use `==` and `!=`. This can cause problems if you start by using `int`s and then later change to `Integer`s, or vice-versa.

In Java 9 and on, the use of `new Integer()` is deprecated because it is so much less efficient than `Integer.valueOf()`. Thus, you should avoid `new Integer()`, `new Double()`, etc. in Java 8 as well. Deprecating something for efficiency reasons is not

something I've seen before (it might have happened, but this is the first time I've been aware of it).

When working with floating-point numbers, you encounter different equivalence problems, not because of Java but because of the nature of floating-point numbers:

```
// operators/DoubleEquivalence.java

public class DoubleEquivalence {
  static void show(String desc, Double n1, Double n2) {
    System.out.println(desc + ":");
    System.out.printf(
      "%e==%e %b %b%n", n1, n2, n1 == n2, n1.equals(n2));
  }
  @SuppressWarnings("deprecation")
  public static void test(double x1, double x2) {
    // x1.equals(x2) // Won't compile
    System.out.printf("%e==%e %b%n", x1, x2, x1 == x2);
    Double d1 = x1;
    Double d2 = x2;
    show("Automatic", d1, d2);
    Double r1 = new Double(x1);
    Double r2 = new Double(x2);
    show("new Double()", r1, r2);
    Double v1 = Double.valueOf(x1);
    Double v2 = Double.valueOf(x2);
    show("Double.valueOf()", v1, v2);
  }
  public static void main(String[] args) {
    test(0, Double.MIN_VALUE);
    System.out.println("-------------------------");
    test(Double.MAX_VALUE,
      Double.MAX_VALUE - Double.MIN_VALUE * 1_000_000);
  }
}
/* Output:
0.000000e+00==4.900000e-324 false
Automatic:
0.000000e+00==4.900000e-324 false false
new Double():
0.000000e+00==4.900000e-324 false false
Double.valueOf():
```

```
0.000000e+00==4.900000e-324 false false
------------------------
1.797693e+308==1.797693e+308 true
Automatic:
1.797693e+308==1.797693e+308 false true
new Double():
1.797693e+308==1.797693e+308 false true
Double.valueOf():
1.797693e+308==1.797693e+308 false true
*/
```

It seems like the comparison of floating point numbers should be very strict—a number that is the tiniest fraction different from another number should still be unequal. This works for the call to `test(0, Double.MIN_VALUE)`, where `Double.MIN_VALUE` is the smallest representable value. (`%e` in the `printf()` call presents the results in exponential notation).

However, this does not hold true for the second `test()` call, where the argument `x2` is the value of `x1` minus one million times `Double.MIN_VALUE`. It appears that `x2` should be significantly different than `x1`, but the two numbers still compare as equal. You encounter this issue in virtually every programming language because when a variable holds a very large number, subtracting a relatively small number won't make a significant difference. This is called a *rounding error* and occurs because the machine cannot hold enough information to represent a tiny change to a large number.

You might be fooled into thinking that using == produces a correct result in this case, but it does not—it is simply comparing references.

Using `equals()` when you're not working with primitives seems like the straightforward answer, but it's not as simple as that. Consider class `ValA`:

```
// operators/EqualsMethod.java
// Default equals() does not compare contents

class ValA {
  int i;
}

class ValB {
  int i;
  // Works for this example, not a complete equals():
  public boolean equals(Object o) {
    ValB rval = (ValB)o;  // Cast o to be a ValB
    return i == rval.i;
  }
}

public class EqualsMethod {
  public static void main(String[] args) {
    ValA va1 = new ValA();
    ValA va2 = new ValA();
    va1.i = va2.i = 100;
    System.out.println(va1.equals(va2));
    ValB vb1 = new ValB();
    ValB vb2 = new ValB();
    vb1.i = vb2.i = 100;
    System.out.println(vb1.equals(vb2));
  }
}
/* Output:
false
true
*/
```

In `main()`, `va1` and `va2` contain identical values for `i`, but the result of comparing them using `equals()` is `false`, which is confusing again. This happens because the default behavior of `equals()` is to compare references. To produce the desired behavior of comparing the contents, you must *override* `equals()` as in class `ValB`. `ValB.equals()` contains only the minimum necessary code to solve the problem for this example, but it is not a proper `equals()`. Note that the standard argument for `equals()` is an `Object` (not a `ValB`), so we must force `o` to be a `ValB` using a *cast*: `(ValB)o` (ordinarily you must check the type first before casting, but we'll skip that

until a later chapter). Then we can compare the two values of `i`, using == because they are primitives.

You won't learn about overriding until the Reuse chapter, and you won't learn the proper way to define `equals()` until the Appendix: Collection Topics, but being aware of the way `equals()` behaves might save you some grief in the meantime.

Most of the standard library classes override `equals()` to compare the contents of objects instead of their references.

Logical Operators

Each of the logical operators AND (&&), OR (||) and NOT (!) produce a `boolean` value of `true` or `false` based on the logical relationship of its arguments. This example uses the relational and logical operators:

```
// operators/Bool.java
// Relational and logical operators
import java.util.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    System.out.println("i = " + i);
    System.out.println("j = " + j);
    System.out.println("i > j is " + (i > j));
    System.out.println("i < j is " + (i < j));
    System.out.println("i >= j is " + (i >= j));
    System.out.println("i <= j is " + (i <= j));
    System.out.println("i == j is " + (i == j));
    System.out.println("i != j is " + (i != j));
    // Treating an int as a boolean is not legal Java:
    //- System.out.println("i && j is " + (i && j));
    //- System.out.println("i || j is " + (i || j));
    //- System.out.println("!i is " + !i);
    System.out.println("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    System.out.println("(i < 10) || (j < 10) is "
```

```
        + ((i < 10) || (j < 10)) );
  }
}
/* Output:
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
*/
```

You can apply AND, OR, or NOT to `boolean` values only. You can't use a non-`boolean` as if it were a `boolean` in a logical expression as you can in C and C++. The failed attempts at doing this are commented out with `//-`. The subsequent expressions, however, produce `boolean` values using relational comparisons, then use logical operations on the results.

Note that a `boolean` value is automatically converted to an appropriate text form if it is used where a `String` is expected.

You can replace the definition for `int` in the preceding program with any other primitive data type except `boolean`.

Short-Circuiting

Logical operators support a phenomenon called "short-circuiting." this means the expression is evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, the latter parts of a logical expression might not be evaluated. Here's a demonstration:

```
// operators/ShortCircuit.java
// Short-circuiting behavior with logical operators

public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
    System.out.println("expression is " + b);
  }
}
/* Output:
test1(0)
result: true
test2(2)
result: false
expression is false
*/
```

Each test performs a comparison against the argument and returns `true` or `false`. It also prints information to show you it's being called. The tests are used in the expression:

```
test1(0) && test2(2) && test3(2)
```

You might naturally expect all three tests to execute, but the output shows otherwise. The first test produces a `true` result, so the expression evaluation continues. However, the second test produces a `false` result. This means the whole expression must be

`false`, so why continue evaluating the rest of the expression? It might be expensive. The reason for short-circuiting, in fact, is that you can get a potential performance increase if all the parts of a logical expression do not need evaluation.

Literals

Ordinarily, when you insert a literal value into a program, the compiler knows exactly what type to make it. When the type is ambiguous, you must guide the compiler by adding some extra information in the form of characters associated with the literal value. The following code shows these characters:

```
// operators/Literals.java

public class Literals {
  public static void main(String[] args) {
    int i1 = 0x2f; // Hexadecimal (lowercase)
    System.out.println(
      "i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Hexadecimal (uppercase)
    System.out.println(
      "i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Octal (leading zero)
    System.out.println(
      "i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // max char hex value
    System.out.println(
      "c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // max byte hex value 0111111;
    System.out.println(
      "b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // max short hex value
    System.out.println(
      "s: " + Integer.toBinaryString(s));
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix (can be confusing)
    long n3 = 200;
    // Java 7 Binary Literals:
    byte blb = (byte)0b00110101;
    System.out.println(
```

```
      "blb: " + Integer.toBinaryString(blb));
    short bls = (short)0B0010111110101111;
    System.out.println(
      "bls: " + Integer.toBinaryString(bls));
    int bli = 0b00101111110101111101011110101111;
    System.out.println(
      "bli: " + Integer.toBinaryString(bli));
    long bll = 0b0010111111010111110101111110101111;
    System.out.println(
      "bll: " + Long.toBinaryString(bll));
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    // (Hex and Octal also work with long)
  }
}
/* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 11111111111111
blb: 110101
bls: 10111110101111
bli: 10111110101111101011111010111
bll: 10111110101111101011111010111
*/
```

A trailing character after a literal value establishes its type. Uppercase or lowercase `L` means `long` (however, using a lowercase `l` is confusing because it can look like the number one). Uppercase or lowercase `F` means `float`. Uppercase or lowercase `D` means `double`.

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading `0x` or `0X` followed by `0-9` or `a-f` either in uppercase or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message. Notice in the preceding code the maximum possible hexadecimal values for `char`, `byte`, and

`short`. If you exceed these, the compiler will automatically make the value an `int` and declare you need a *narrowing cast* for the assignment (casts are defined later in this chapter). You'll know you've stepped over the line.

Octal (base 8) is denoted by a leading zero in the number and digits from 0-7.

Java 7 introduced binary literals, denoted by a leading `0b` or `0B`, which can initialize all integral types.

When working with integral types, it's useful to display the binary form of the results. This is easily accomplished with the `static toBinaryString()` methods from the `Integer` and `Long` classes. Notice that when passing smaller types to `Integer.toBinaryString()`, the type is automatically converted to an `int`.

Underscores in Literals

There's a thoughtful addition in Java 7: you can include underscores in numeric literals so the results are easier to read. This is especially helpful for grouping digits in large values:

```
// operators/Underscores.java

public class Underscores {
  public static void main(String[] args) {
    double d = 341_435_936.445_667;
    System.out.println(d);
    int bin = 0b0010_1111_1010_1111_1010_1111_1010_1111;
    System.out.println(Integer.toBinaryString(bin));
    System.out.printf("%x%n", bin);              // [1]
    long hex = 0x7f_e9_b7_aa;
    System.out.printf("%x%n", hex);
  }
}
/* Output:
3.41435936445667E8
10111110101111101011111010111
2fafafaf
7fe9b7aa
*/
```

There are (reasonable) rules:

1. Single underscores only—you can't double them up.
2. No underscores at the beginning or end of a number.
3. No underscores around suffixes like `F`, `D` or `L`.
4. No around binary or hex identifiers `b` and `x`.

- **[1]**: Notice the use of `%n`. If you're familiar with C-style languages, you're probably used to seeing `\n` to represent a line ending. The problem with that is it gives you a "Unix style" line ending. If you are on Windows, you must specify `\r\n` instead. This difference is a needless hassle; the programming language should take care of it for you. That's what Java has achieved with `%n`, which always produces the appropriate line ending for the platform it's running on—but only when you're using `System.out.printf()` or `System.out.format()`. For `System.out.println()` you must still use `\n`; if you use `%n`, `println()` will simply emit `%n` and not a newline.

Exponential Notation

Exponents use a notation I've always found rather dismaying:

```
// operators/Exponents.java
// "e" means "10 to the power."

public class Exponents {
  public static void main(String[] args) {
    // Uppercase and lowercase 'e' are the same:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' is optional
    double expDouble2 = 47e47; // Automatically double
    System.out.println(expDouble);
  }
}
/* Output:
1.39E-43
4.7E48
*/
```

In science and engineering, `e` refers to the base of natural logarithms, approximately 2.718. (A more precise `double` value is available in Java as `Math.E`.) This is used in exponentiation expressions such as 1.39 x e-43, which means 1.39 x 2.718-43. However, when the FORTRAN programming language was invented, they decided that `e` would mean "ten to the power," an odd decision because FORTRAN was designed for science and engineering, and one would think its designers would be sensitive about introducing such an ambiguity.[58] At any rate, this custom was followed in C, C++ and now Java. So if you're used to thinking in terms of `e` as the base of natural logarithms, you must do a mental translation when you see an expression such as `1.39 e-43f` in Java; it means 1.39 x 10-43.

Note you don't need the trailing character when the compiler can figure out the appropriate type. With

```
long n3 = 200;
```

there's no ambiguity, so an `L` after the 200 is superfluous. However, with

```
float f4 = 1e-43f; // 10 to the power
```

the compiler normally takes exponential numbers as doubles, so without the trailing `f`, it will give you an error declaring you must use a cast to convert `double` to `float`.

Bitwise Operators

The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform Boolean algebra on the corresponding bits in the two arguments to produce the result.

The bitwise operators come from C's low-level orientation, where you often manipulate hardware directly and must set the bits in hardware registers. Java was originally

[58] John Kirkham writes, "I started computing in 1962 using FORTRAN II on an IBM 1620. At that time, and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units that used 5 bit Baudot code, which had no lowercase capability. The `E` in the exponential notation was also always uppercase and was never confused with the natural logarithm base `e`, which is always lowercase. The `E` simply stood for exponential, which was for the base of the number system used—usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lowercase `e` was in the late 1970s and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase."

designed to be embedded in TV control boxes, so this low-level orientation still made sense. However, you probably won't use the bitwise operators much.

The bitwise AND operator (`&`) produces a one in the output bit if both input bits are one; otherwise, it produces a zero. The bitwise OR operator (`|`) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. The bitwise EXCLUSIVE OR, or XOR (`^`), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the *ones complement* operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces the opposite of the input bit—a one if the input bit is zero, a zero if the input bit is one.

The bitwise operators and logical operators use the same characters, so a mnemonic device helps you remember the meanings: Because bits are "small," there is only one character in the bitwise operators.

Bitwise operators can be combined with the = sign to unite the operation and assignment: `&=`, `|=` and `^=` are all legitimate. (Because ~ is a unary operator, it cannot be combined with the = sign.)

The `boolean` type is treated as a one-bit value, so it is somewhat different. You can perform a bitwise AND, OR, and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For `booleans`, the bitwise operators have the same effect as the logical operators except they do not short circuit. Also, bitwise operations on `booleans` include an XOR logical operator that is not included under the list of "logical" operators. You cannot use `booleans` in shift expressions, which are described next.

Shift Operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types. The left-shift operator (`<<`) produces the operand to the left of the operator after it is shifted to the left by the number of bits specified to the right of the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (`>>`) produces the operand to the left of the operator after it is shifted to the right by the number of bits specified to the right of the operator. The signed right shift `>>` uses *sign extension*: If the value is positive, zeroes are inserted at the higher-order

bits; if the value is negative, ones are inserted at the higher-order bits. Java has also added the unsigned right shift `>>>`, which uses *zero extension*: Regardless of the sign, zeroes are inserted at the higher-order bits. This operator does not exist in C or C++.

If you shift a `char`, `byte`, or `short`, it is promoted to `int` before the shift takes place, and the result is an `int`. Only the five low-order bits of the right-hand side are used. This prevents you from shifting more than the number of bits in an `int`. If you're operating on a `long`, you'll get a `long` result. Only the six low-order bits of the right-hand side are used, so you can't shift more than the number of bits in a `long`.

Shifts can be combined with the equal sign (`<<=` or `>>=` or `>>>=`). The lvalue is replaced by the lvalue shifted by the rvalue. There is a problem, however, with the unsigned right shift combined with assignment. If you use it with `byte` or `short`, you don't get the correct results. Instead, these are promoted to `int` and right shifted, but then truncated as they are assigned back into their variables, so you get `-1` in those cases. Here's a demonstration:

```
// operators/URShift.java
// Test of unsigned right shift

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    System.out.println(Integer.toBinaryString(i));
    i >>>= 10;
    System.out.println(Integer.toBinaryString(i));
    long l = -1;
    System.out.println(Long.toBinaryString(l));
    l >>>= 10;
    System.out.println(Long.toBinaryString(l));
    short s = -1;
    System.out.println(Integer.toBinaryString(s));
    s >>>= 10;
    System.out.println(Integer.toBinaryString(s));
    byte b = -1;
    System.out.println(Integer.toBinaryString(b));
    b >>>= 10;
    System.out.println(Integer.toBinaryString(b));
    b = -1;
    System.out.println(Integer.toBinaryString(b));
    System.out.println(Integer.toBinaryString(b>>>10));
```

```
  }
}
/* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*/
```

In the last shift, the resulting value is not assigned back into `b`, but is printed directly, so the correct behavior occurs.

Here's an example that exercises all the operators involving bits:

```
// operators/BitManipulation.java
// Using the bitwise operators
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt();
    int j = rand.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
    int maxpos = 2147483647;
    printBinaryInt("maxpos", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("maxneg", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
```

```
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("maxpos", ll);
    long lln = -9223372036854775808L;
    printBinaryLong("maxneg", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary:\n   " +
      Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary:\n    " +
      Long.toBinaryString(l));
  }
}
/* Output: (First 32 Lines)
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
```

```
   1
maxpos, int: 2147483647, binary:
   11111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: -1172028779, binary:
   10111010001001000100001010010101
~i, int: 1172028778, binary:
   01000101110110111011110101101010
-i, int: 1172028779, binary:
   01000101110110111011110101101011
j, int: 1717241110, binary:
   01100110010110110000010100010110
i & j, int: 570425364, binary:
   00010000000000000000000000010100
i | j, int: -25213033, binary:
   11111110011111111010001111010111
i ^ j, int: -595638397, binary:
   11011100011111111010001111010011
i << 5, int: 1149784736, binary:
   01000100100010000101001010100000
i >> 5, int: -36625900, binary:
   11111101111010001001000100010100
(~i) >> 5, int: 36625899, binary:
   00000101110110111011101011
i >>> 5, int: 97591828, binary:
   00000101110110111011101011
(~i) >>> 5, int: 36625899, binary:
   00000101110110111011101011
                  ...
*/
```

The two methods at the end, `printBinaryInt()` and `printBinaryLong()`, take an `int` or a `long`, respectively, and display it in binary format along with a descriptive `String`. As well as demonstrating the effect of all the bitwise operators for `int` and `long`, this example also shows the minimum, maximum, +1, and -1 values for `int` and `long` so you see what they look like. Note that the high bit represents the sign: 0 means positive and 1 means negative. The output for the `int` portion is displayed above.

The binary representation of the numbers is called *signed twos complement.*

Ternary `if-else` Operator

The *ternary* operator, also called the *conditional* operator, is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary `if-else` statement that you'll see in the next chapter. The expression is of the form:

```
boolean-exp ? value0 : value1
```

If *boolean-exp* evaluates to `true`, *value0* is evaluated, and its result becomes the value produced by the operator. If *boolean-exp* is `false`, *value1* is evaluated and its result becomes the value produced by the operator.

You can also use an ordinary `if-else` statement (described later), but the ternary operator is much terser. Although C (where this operator originated) prides itself on being a terse language, and the ternary operator might have been introduced partly for efficiency, be somewhat wary of using it on an everyday basis—it's easy to produce unreadable code.

The ternary operator is different from `if-else` because it produces a value. Here's an example comparing the two:

```
// operators/TernaryIfElse.java

public class TernaryIfElse {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int standardIfElse(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    System.out.println(ternary(9));
    System.out.println(ternary(10));
    System.out.println(standardIfElse(9));
    System.out.println(standardIfElse(10));
  }
}
```

```
/* Output:
900
100
900
100
*/
```

The code in `ternary()` is more compact than what you'd write without the ternary operator, in `standardIfElse()`. However, `standardIfElse()` is easier to understand, and doesn't require a lot more typing. Ponder your reasons when choosing the ternary operator—it's primarily warranted when you're setting a variable to one of two values.

`String` Operator + and +=

There's one special usage of an operator in Java: The + and += operators can concatenate `String`s, as you've already seen. It seems a natural use of these operators even though it doesn't fit with the traditional way they are used.

This capability seemed like a good idea in C++, so *operator overloading* was added to C++ to allow the C++ programmer to add meanings to almost any operator. Unfortunately, operator overloading combined with some of the other restrictions in C++ turns out to be a fairly complicated feature for programmers to design into their classes. Although operator overloading would have been much simpler to implement in Java than it was in C++ (as demonstrated by the C# language, which *does* have straightforward operator overloading), this feature was still considered too complex, so Java programmers cannot implement their own overloaded operators like C++ and C# programmers can.

If an expression begins with a `String`, all operands that follow must be `String`s (remember that the compiler automatically turns a double-quoted sequence of characters into a `String`):

```
// operators/StringOperators.java

public class StringOperators {
  public static void main(String[] args) {
    int x = 0, y = 1, z = 2;
    String s = "x, y, z ";
    System.out.println(s + x + y + z);
    // Converts x to a String:
    System.out.println(x + " " + s);
    s += "(summed) = "; // Concatenation operator
    System.out.println(s + (x + y + z));
    // Shorthand for Integer.toString():
    System.out.println("" + x);
  }
}
/* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*/
```

Note that the output from the first print statement is `012` instead of just `3`, which you'd get if it was summing the integers. This is because the Java compiler converts `x`, `y`, and `z` into their `String` representations and concatenates those `Strings`, instead of adding them together first. The second print statement converts the leading variable into a `String`, so the `String` conversion does not depend on what comes first. Finally, you see the `+=` operator to append a `String` to `s`, and parentheses to control the order of evaluation of the expression so the `int`s are actually summed before they are displayed.

Notice the last example in `main()`: you sometimes see an empty `String` followed by a `+` and a primitive as a way to perform the conversion without calling the more cumbersome explicit method (`Integer.toString()`, here).

Common Pitfalls When Using Operators

One of the pitfalls when using operators is attempting to leave out the parentheses when you are even the least bit uncertain about how an expression will evaluate.

This is still true in Java.

An extremely common error in C and C++ looks like this:

```
while(x = y) {
  // ...
}
```

The programmer was clearly trying to test for equivalence (==) rather than do an assignment. In C and C++ the result of this assignment will always be `true` if `y` is nonzero, and you'll probably get an infinite loop. In Java, the result of this expression is not a `boolean`, but the compiler expects a `boolean` and won't convert from an `int`, so it will conveniently give you a compile-time error and catch the problem before you ever try to run the program. So the pitfall never happens in Java. (The only time you won't get a compile-time error is when `x` and `y` are `boolean`, in which case `x = y` is a legal expression, and in the preceding example, probably an error.)

A similar problem in C and C++ is using bitwise AND and OR instead of the logical versions. Bitwise AND and OR use one of the characters (`&` or `|`) while logical AND and OR use two (`&&` and `||`). Just as with = and ==, it's easy to type just one character instead of two. In Java, the compiler again prevents this, because it won't let you cavalierly use one type where it doesn't belong.

Casting Operators

The word *cast* is used in the sense of "casting into a mold." Java will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating point variable, the compiler will automatically convert the `int` to a `float`. Casting makes this type conversion explicit, or forces it when it wouldn't normally happen.

To perform a cast, put the desired data type inside parentheses to the left of any value, as seen here:

```
// operators/Casting.java

public class Casting {
  public static void main(String[] args) {
    int i = 200;
    long lng = (long)i;
    lng = i; // "Widening," so a cast is not required
    long lng2 = (long)200;
    lng2 = 200;
    // A "narrowing conversion":
    i = (int)lng2; // Cast required
  }
}
```

Thus, you can cast a numeric value as well as a variable. Casts may be superfluous; for example, the compiler will automatically promote an `int` value to a `long` when necessary. However, you are allowed to use superfluous casts to make a point or to clarify your code. In other situations, a cast might be essential just to get the code to compile.

In C and C++, casting can cause some headaches. In Java, casting is safe, with the exception that when you perform a so-called *narrowing conversion* (that is, when you go from a data type that can hold more information to one that doesn't hold as much), you run the risk of losing information. Here the compiler forces you to use a cast, in effect saying, "This can be a dangerous thing to do—if you want me to do it anyway you must make the cast explicit." With a *widening conversion* an explicit cast is not needed, because the new type will more than hold the information from the old type so no information is ever lost.

Java can cast any primitive type to any other primitive type, except for `boolean`, which doesn't allow any casting at all. Class types do not allow casting. To convert one to the other, there must be special methods. (You'll find out later that objects can be cast within a *family* of types; an `Oak` can be cast to a `Tree` and vice versa, but not to a foreign type such as a `Rock`.)

Truncation and Rounding

When you are performing narrowing conversions, you must pay attention to issues of truncation and rounding. For example, if you cast from a floating point value to

an integral value, what does Java do? For example, if you cast the value 29.7 to an `int`, is the resulting value 30 or 29? The answer is seen here:

```
// operators/CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?

public class CastingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    System.out.println("(int)above: " + (int)above);
    System.out.println("(int)below: " + (int)below);
    System.out.println("(int)fabove: " + (int)fabove);
    System.out.println("(int)fbelow: " + (int)fbelow);
  }
}
/* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*/
```

So the answer is that casting from a `float` or `double` to an integral value always truncates the number. If instead you want the result rounded, use the `round()` methods in `java.lang.Math`:

```
// operators/RoundingNumbers.java
// Rounding floats and doubles

public class RoundingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    System.out.println(
      "Math.round(above): " + Math.round(above));
    System.out.println(
      "Math.round(below): " + Math.round(below));
    System.out.println(
      "Math.round(fabove): " + Math.round(fabove));
    System.out.println(
```

```
      "Math.round(fbelow): " + Math.round(fbelow));
  }
}
/* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*/
```

`round()` is part of `java.lang` so you don't need an extra import to use it.

Promotion

You'll discover that if you perform any mathematical or bitwise operations on primitive data types smaller than an `int` (that is, `char`, `byte`, or `short`), those values are promoted to `int` before performing the operations, and the resulting value is of type `int`. To assign back into the smaller type, you use a cast. (And, because you're assigning back into a smaller type, you might be losing information.) In general, the largest data type in an expression is the one that determines the size of the result of that expression. If you multiply a `float` and a `double`, the result is `double`. If you add an `int` and a `long`, the result is `long`.

Java Has No "sizeof"

In C and C++, the `sizeof()` operator tells you the number of bytes allocated for data items. The most compelling reason for `sizeof()` in C and C++ is for portability. Different data types might be different sizes on different machines, so the programmer must discover how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine. As you might imagine, portability is a huge headache for C and C++ programmers.

Java does not need a `sizeof()` operator for this purpose, because all the data types are the same size on all machines. You do not need to think about portability on this level—it is designed into the language.

A Compendium of Operators

The following example shows which primitive data types can be used with particular operators. Basically, it is the same example repeated over and over, but using different primitive data types. The file will compile without error because the lines that fail are commented out with a `//-`.

```
// operators/AllOps.java
// Tests all operators on all primitive data types
// to show which ones are accepted by the Java compiler

public class AllOps {
  // To accept the results of a Boolean test:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Arithmetic operators:
    //- x = x * y;
    //- x = x / y;
    //- x = x % y;
    //- x = x + y;
    //- x = x - y;
    //- x++;
    //- x--;
    //- x = +y;
    //- x = -y;
    // Relational and logical:
    //- f(x > y);
    //- f(x >= y);
    //- f(x < y);
    //- f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Bitwise operators:
    //- x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //- x = x << 1;
```

```
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    //- x += y;
    //- x -= y;
    //- x *= y;
    //- x /= y;
    //- x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- char c = (char)x;
    //- byte b = (byte)x;
    //- short s = (short)x;
    //- int i = (int)x;
    //- long l = (long)x;
    //- float f = (float)x;
    //- double d = (double)x;
  }
  void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char) + y;
    x = (char) - y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
```

```
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte) + y;
    x = (byte) - y;
```

```
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void shortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
```

```
x = (short)(x / y);
x = (short)(x % y);
x = (short)(x + y);
x = (short)(x - y);
x++;
x--;
x = (short) + y;
x = (short) - y;
// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//- f(!x);
//- f(x && y);
//- f(x || y);
// Bitwise operators:
x = (short) ~ y;
x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
x = (short)(x >>> 1);
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//- boolean bl = (boolean)x;
char c = (char)x;
byte b = (byte)x;
```

```
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
```

```
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void longTest(long x, long y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
```

```
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
```

```
    // Bitwise operators:
    //- x = ~y;
    //- x = x & y;
    //- x = x | y;
    //- x = x ^ y;
    //- x = x << 1;
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    //- x &= y;
    //- x ^= y;
    //- x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
```

```
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    //- x = ~y;
    //- x = x & y;
    //- x = x | y;
    //- x = x ^ y;
    //- x = x << 1;
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    //- x &= y;
    //- x ^= y;
    //- x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
}
```

Note that `boolean` is limited. You can assign it the values `true` and `false`, and you can test it for truth or falsehood, but you cannot add `boolean`s or perform any other type of operation on them.

In `char`, `byte`, and `short`, you see the effect of promotion with the arithmetic operators. Each arithmetic operation on any of those types produces an `int` result, which must be explicitly cast back to the original type (a narrowing conversion that might lose information) to assign back to that type. With `int` values, however, you do not need a cast, because everything is already an `int`. Don't be lulled into thinking everything is safe, though. If you multiply two `int`s that are big enough, you'll overflow the result. The following example demonstrates this:

```
// operators/Overflow.java
// Surprise! Java lets you overflow

public class Overflow {
  public static void main(String[] args) {
    int big = Integer.MAX_VALUE;
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
  }
}
/* Output:
big = 2147483647
bigger = -4
*/
```

You get no errors or warnings from the compiler, and no runtime exceptions.

Compound assignments do *not* require casts for `char`, `byte`, or `short`, even though they are performing promotions that have the same results as the direct arithmetic operations. On the other hand, the lack of a cast certainly simplifies the code.

Except for `boolean`, any primitive type can be cast to any other primitive type. Again, you must be aware of the effect of a narrowing conversion when casting to a smaller type; otherwise, you might unknowingly lose information during the cast.

Summary

If you've had experience with any languages that use C-like syntax, you see that the operators in Java are so similar there is virtually no learning curve.